软件测试

- 软件测试的目的和原则
- 软件测试用例设计
- 软件测试策略
- 软件测试种类
- 程序调试

软件测试的目的和原则

- 软件测试的目的
- 软件测试的原则
- 软件测试的对象
- 测试信息流
- 测试与软件开发各阶段的关系

软件测试的目的

- 基于不同的立场，存在着两种完全不同的测试目的。
- 从用户的角度出发，普遍希望通过软件测试暴露软件中隐藏的错误和缺陷，以考虑是否可接受该产品。
- 从软件开发者的角度出发，则希望测试成为表明软件产品中不存在错误的过程，验证该软件已正确地实现了用户的要求，确立人们对软件质量的信心。

*Myers*软件测试目的

(1) 测试是程序的执行过程，目的在于发现错误；

(2) 一个好的测试用例在于能发现至今未发现的错误；

(3) 一个成功的测试是发现了至今未发现的错误的测试。

- 换言之，测试的目的是
 - 想以最少的时间和人力，系统地找出软件中潜在的各种错误和缺陷。如果我们成功地实施了测试，我们就能够发现软件中的错误。
 - 测试的附带收获是，它能够证明软件的功能和性能与需求说明相符合。
 - 实施测试收集到的测试结果数据为可靠性分析提供了依据。
 - 测试不能表明软件中不存在错误，它只能说明软件中存在错误。



软件测试的原则

1. 应当把“尽早地和不断地进行软件测试”作为软件开发者的座右铭。
2. 测试用例应由测试输入数据和对应的预期输出结果这两部分组成。
3. 程序员应避免检查自己的程序。
4. 在设计测试用例时，应包括合理的输入条件和不合理的输入条件。

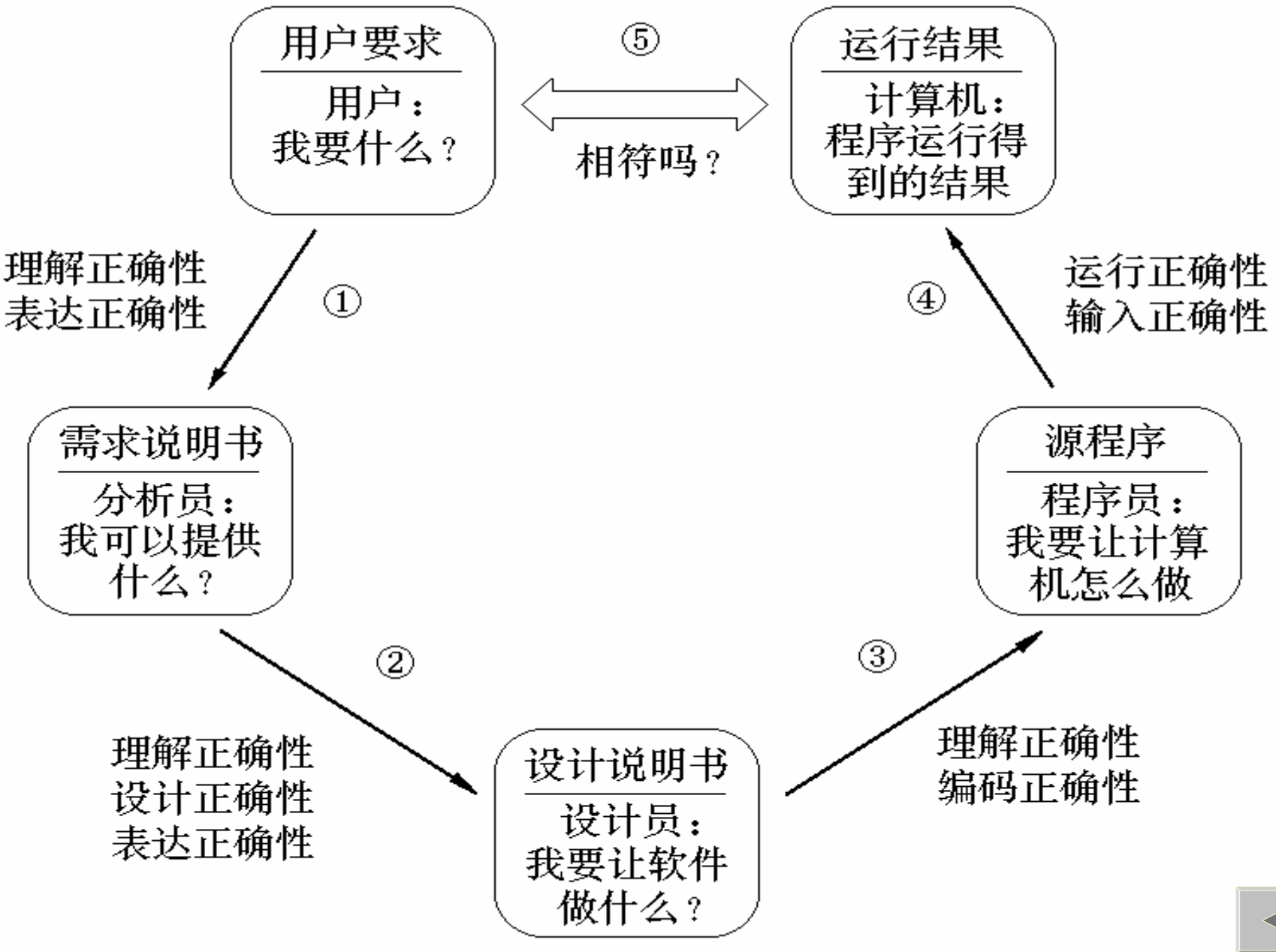
5. 充分注意测试中的群集现象。
经验表明，测试后程序中残存的错误数目与该程序中已发现的错误数目成正比。
6. 严格执行测试计划，排除测试的随意性。
7. 应当对每一个测试结果做全面检查。
8. 妥善保存测试计划，测试用例，出错统计和最终分析报告，为维护提供方便。



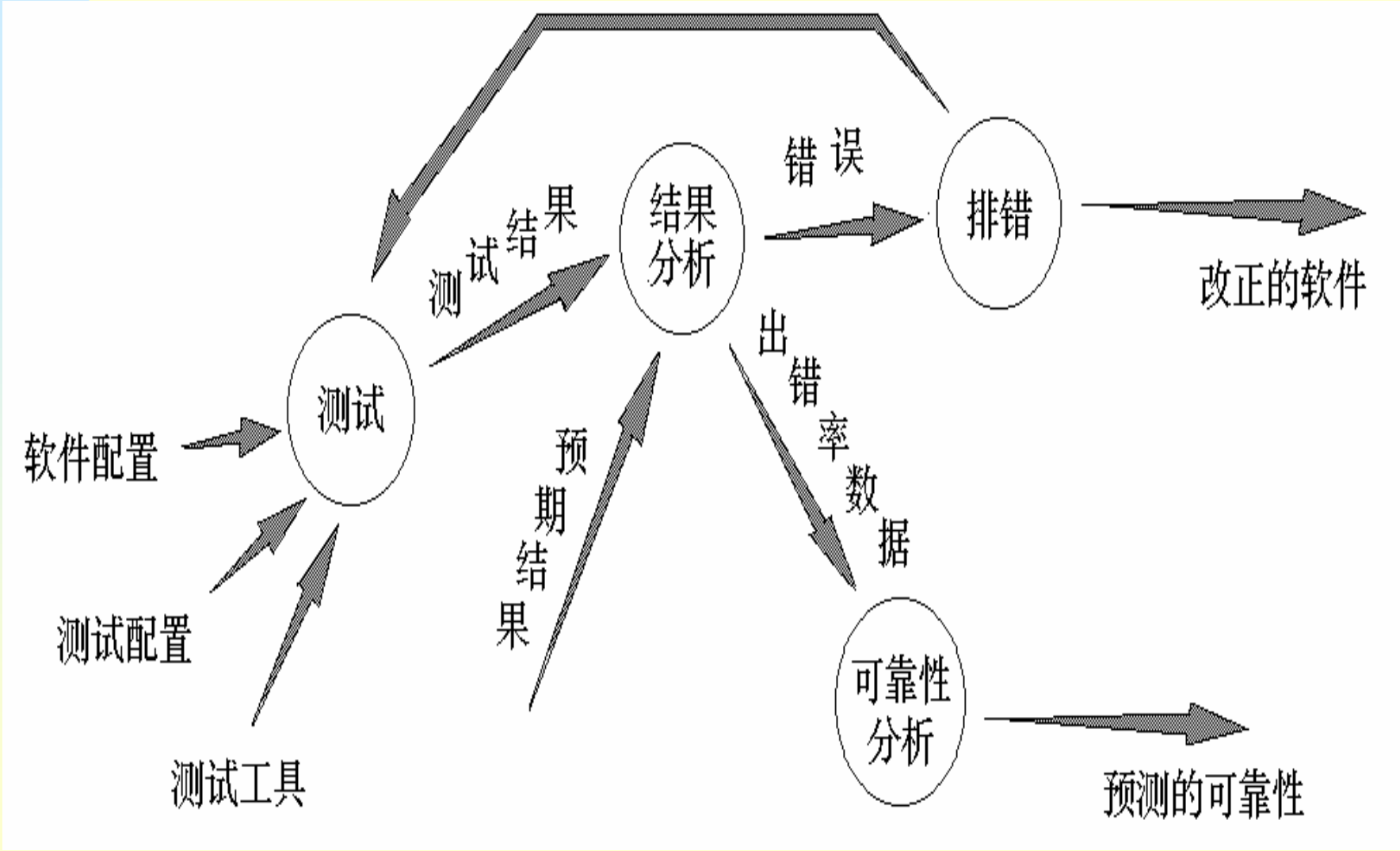
软件测试的对象

- 软件测试并不等于程序测试。软件测试应贯穿于软件定义与开发的整个期间。
- 需求分析、概要设计、详细设计以及程序编码等各阶段所得到的文档，包括需求规格说明、概要设计规格说明、详细设计规格说明以及源程序，都应成为软件测试的对象。

- 为把握软件开发各个环节的正确性，需要进行各种**确认**和**验证**工作。
- **确认(*Validation*)**，是一系列的活动和过程，目的是想证实在一个给定的外部环境中软件的逻辑正确性。
 - 需求规格说明确认
 - 程序确认 (静态确认、动态确认)
- **验证(*Verification*)**，试图证明在软件生存期各个阶段，以及阶段间的逻辑协调性、完备性和正确性。


用户要求
用户：
我要什么？
⑤
相符吗？
运行结果
计算机：
程序运行得
到的结果
理解正确性
表达正确性
①
④
运行正确性
输入正确性
需求说明书
分析员：
我可以提供
什么？
源程序
程序员：
我要让计算
机怎么做
②
③
理解正确性
设计正确性
表达正确性
设计说明书
设计员：
我要让软件
做什么？
理解正确性
编码正确性

测试信息流


测试
结果分析
排错
改正的软件
错误
测试结果
软件配置
测试配置
测试工具
预期结果
出错率数据
可靠性分析
预测的可靠性

测试信息流

- **软件配置**：软件需求规格说明、软件设计规格说明、源代码等；
- **测试配置**：测试计划、测试用例、测试程序等；
- **测试工具**：测试数据自动生成程序、静态分析程序、动态分析程序、测试结果分析程序、以及驱动测试的测试数据库等等。

- 测试结果分析：比较实测结果与预期结果，评价错误是否发生。
- 排错(调试)：对已经发现的错误进行错误定位和确定出错性质，并改正这些错误，同时修改相关的文档。
- 修正后的文档再测试：直到通过测试为止。

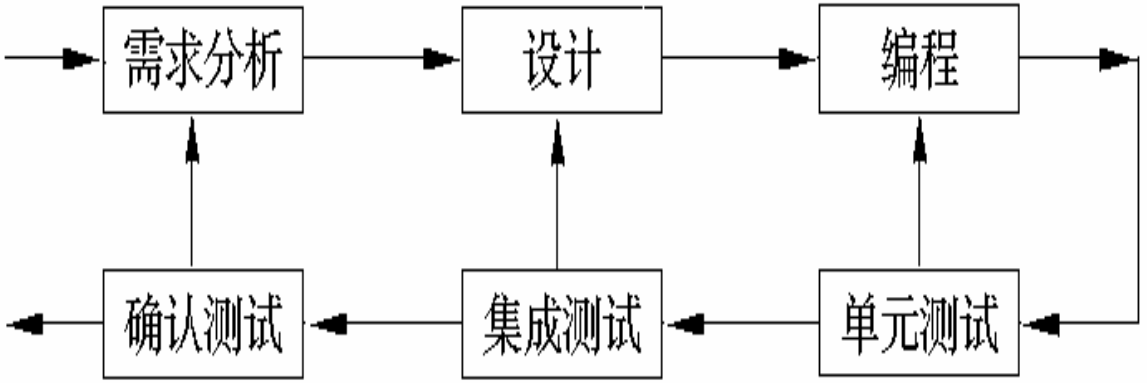
- 通过收集和分析测试结果数据，对软件建立可靠性模型
- 利用可靠性分析，评价软件质量：
 - 软件的质量和可靠性达到可以接受的程度；
 - 所做的测试不足以发现严重的错误；
- 如果测试发现不了错误，可以肯定，测试配置考虑得不够细致充分，错误仍然潜伏在软件中。

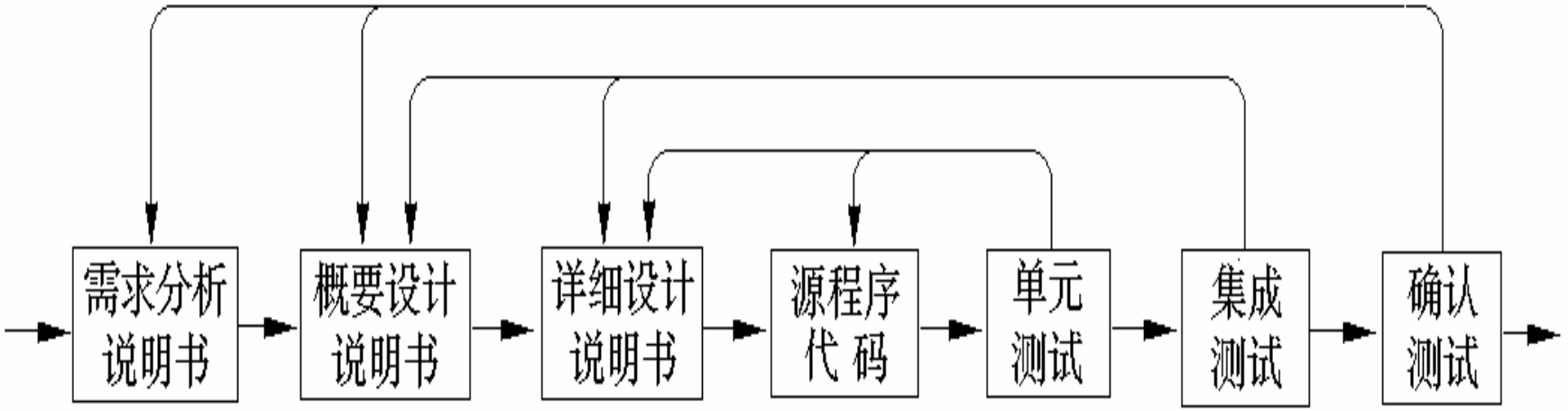


测试与软件开发各阶段的关系

- 软件开发过程是一个自顶向下，逐步细化的过程
- 软件计划阶段定义软件作用域
- 软件需求分析建立软件信息域、功能和性能需求、约束等
- 软件设计
- 把设计用某种程序设计语言转换成程序代码

- 测试过程是依相反顺序安排的自底向上，逐步集成的过程。





测试用例设计

- 两种常用的测试方法
 - 黑盒测试
 - 白盒测试

黑盒测试

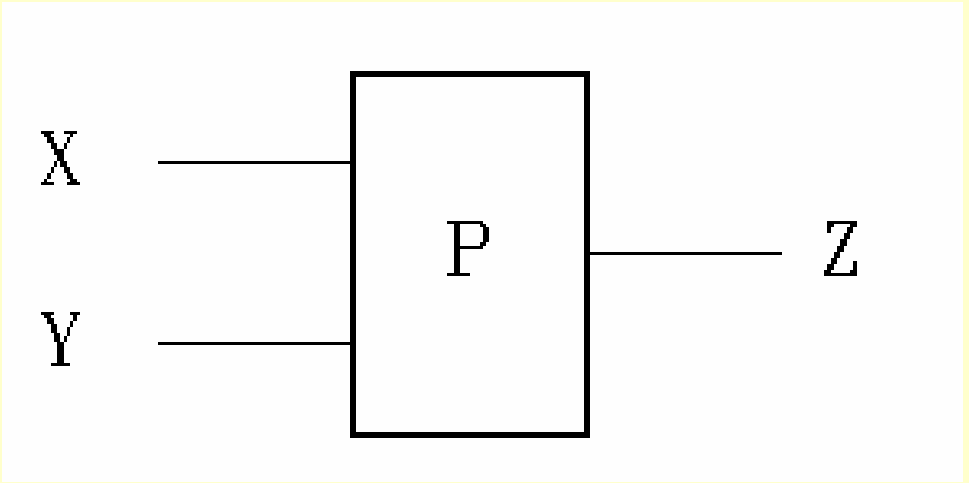
- 这种方法是把测试对象看做一个黑盒子，测试人员完全不考虑程序内部的逻辑结构和内部特性，只依据程序的需求规格说明书，检查程序的功能是否符合它的功能说明。
- 黑盒测试又叫做功能测试或数据驱动测试。

- 黑盒测试方法是在程序接口上进行测试，主要是为了发现以下错误:
 - 是否有不正确或遗漏了的功能?
 - 在接口上，输入能否正确地接受?能否输出正确的结果?
 - 是否有数据结构错误或外部信息(例如数据文件)访问错误?
 - 性能上是否能够满足要求?
 - 是否有初始化或终止性错误?

- 用黑盒测试发现程序中的错误，必须在所有可能的输入条件和输出条件中确定测试数据，来检查程序是否都能产生正确的输出。
- 但这是不可能的。

- 假设一个程序*P*有输入量*X*和*Y*及输出量*Z*。在字长为32位的计算机上运行。若*X*、*Y*取整数，按黑盒方法进行穷举测试：
- 可能采用的测试数据组

$$2^{32} \times 2^{32} = 2^{64}$$



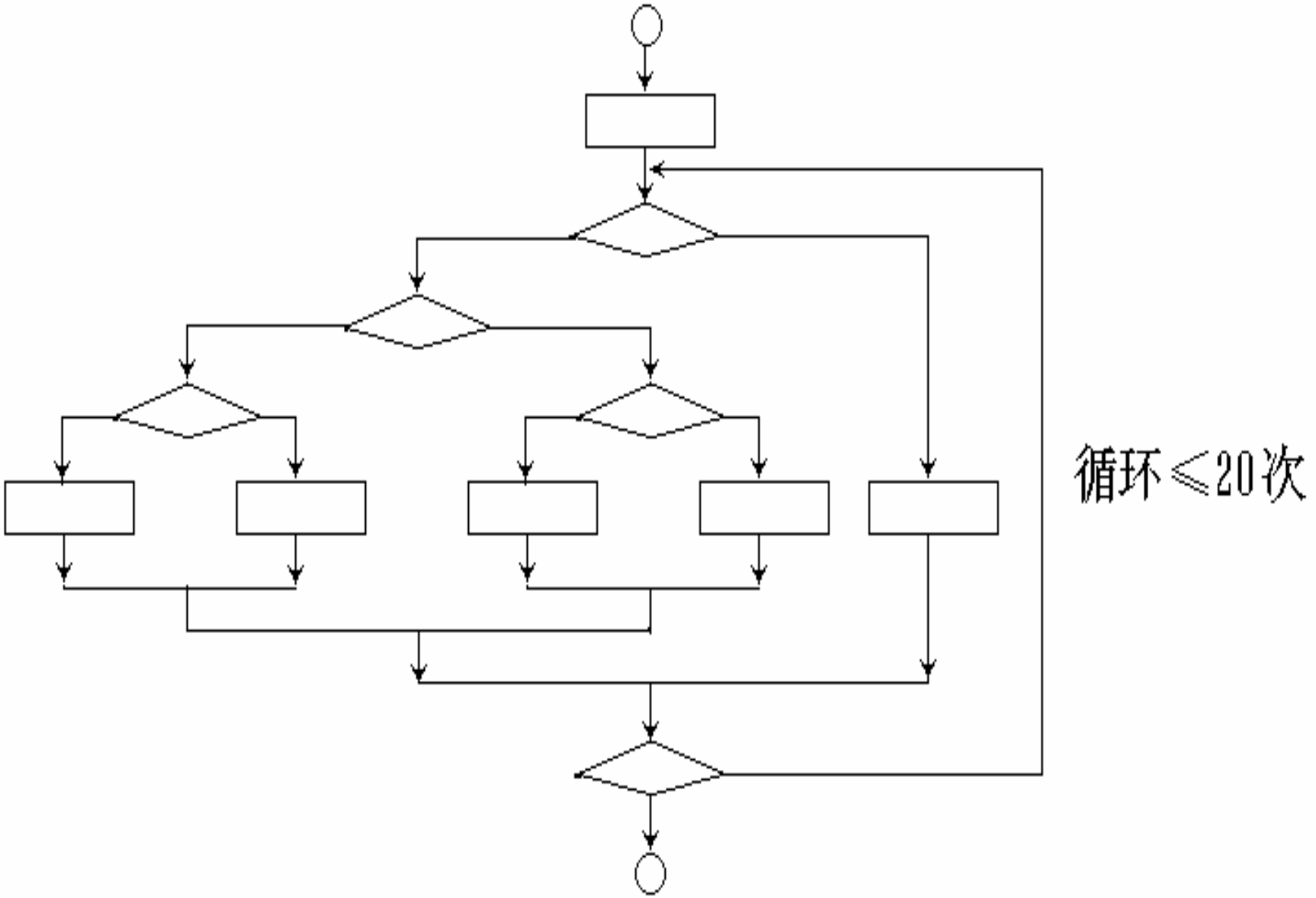
- 如果测试一组数据需要1毫秒，一年工作*365*× *24*小时，完成所有测试需5亿年。

白盒测试

- 此方法把测试对象看做一个透明的盒子，它允许测试人员利用程序内部的逻辑结构及有关信息，设计或选择测试用例，对程序所有逻辑路径进行测试。
- 通过在不同点检查程序的状态，确定实际的状态是否与预期的状态一致。因此白盒测试又称为结构测试或逻辑驱动测试。

- 软件人员使用白盒测试方法，主要想对程序模块进行如下的检查：
 - 对程序模块的所有独立的执行路径至少测试一次；
 - 对所有的逻辑判定，取“真”与取“假”的两种情况都至少测试一次；
 - 在循环的边界和运行界限内执行循环体；
 - 测试内部数据结构的有效性，等。

- 对一个具有多重选择和循环嵌套的程序，不同的路径数目可能是天文数字。给出一个小程序的流程图，它包括了一个执行*20*次的循环。
- 包含的不同执行路径数达5^{20}条，对每一条路径进行测试需要1毫秒，假定一年工作*365* × *24*小时，要想把所有路径测试完，需3170年。

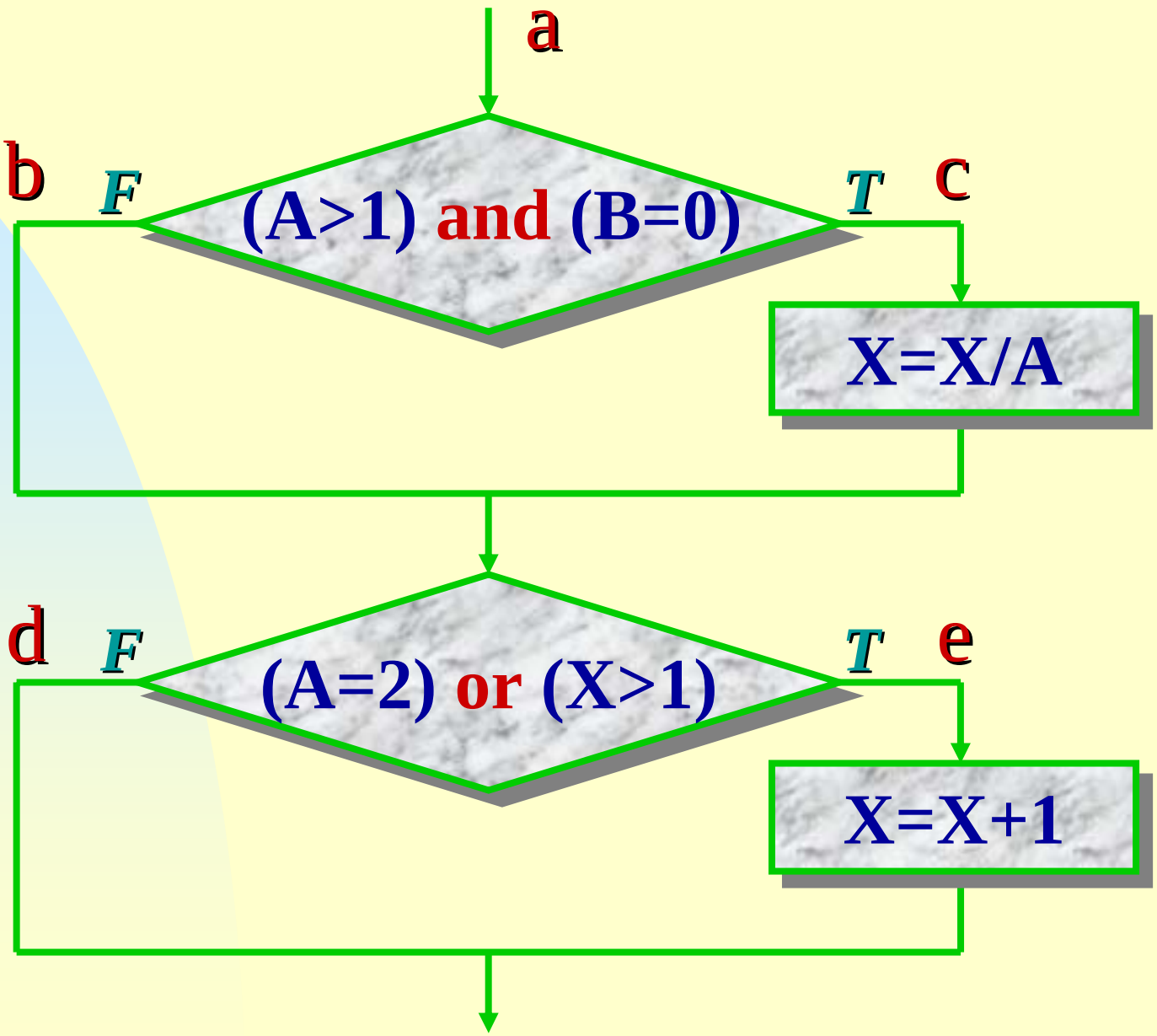

循环≤20次

逻辑覆盖

逻辑覆盖是以程序内部的逻辑结构为基础的设计测试用例的技术。它属白盒测试。

- 语句覆盖
- 判定覆盖
- 条件覆盖
- 判定—条件覆盖
- 条件组合覆盖
- 路径覆盖。




a
b
F
(A>1) and (B=0)
T
c
X=X/A
d
F
(A=2) or (X>1)
T
e
X=X+1

L1 (a → c → e)

= {(A>1) and (B=0)} and

{(A=2) or (X/A>1)}

= (A>1) and (B=0) and (A=2) or

(A>1) and (B=0) and (X/A>1)

= (A=2) and (B=0) or

(A>1) and (B=0) and (X/A>1)

L2 (a→ b → d)

= not{(A>1) and (B=0)} and

not{(A=2) or (X>1)}

= { not (A>1) or not (B=0) } and

{ not (A=2) and not (X>1) }

= not (A>1) and not (A=2) and not (X>1)

or

not (B=0) and not (A=2) and not (X>1)

L3 (a→ b→ e)

= not {(A>1) and (B=0)} and

{(A=2) or (X>1)}

= { not (A>1) or not (B=0)} and

{(A=2) or (X>1)}

= not (A>1) and (A=2) or

not (A>1) and (X>1) or

not (B=0) and (A=2) or

not (B=0) and (X>1)

L4 (a→ c → d)

= {(A>1) and (B=0)} and

not {(A=2) or (X/A>1)}

= (A>1) and (B=0) and not (A=2) and

not (X/A>1)

语句覆盖

- 语句覆盖就是设计若干个测试用例，运行被测程序，使得每一可执行语句至少执行一次。
- 在图例中，正好所有的可执行语句都在路径L1上，所以选择路径 L1设计测试用例，就可以覆盖所有的可执行语句。

- 测试用例的设计格式如下
 【输入的(A, B, X)，输出的(A, B, X)】
- 为图例设计满足语句覆盖的测试用例是:
 【(2, 0, 4)，(2, 0, 3)】

覆盖 ace 【L1】

(A=2) and (B=0) or

(A>1) and (B=0) and (X/A>1)

判定覆盖

- 判定覆盖就是设计若干个测试用例，运行被测程序，使得程序中每个判断的取真分支和取假分支至少经历一次。
- 判定覆盖又称为分支覆盖。
- 对于图例，如果选择路径L1和L2，就可得满足要求的测试用例：

- 【(2, 0, 4)，(2, 0, 3)】覆盖 ace
 【L1】
 【(1, 1, 1)，(1, 1, 1)】覆盖 abd
 【L2】

(A=2) and (B=0) or

(A>1) and (B=0) and (X/A>1)

not (A>1) and not (A=2) and not (X>1)

or

not (B=0) and not (A=2) and not (X>1)

- 如果选择路径L3和L4，还可得另一组可用的测试用例:

 【(2, 1, 1)，(2, 1, 2)】覆盖 abe

 【L3】

 【(3, 0, 3)，(3, 1, 1)】覆盖 acd

 【L4】

not (A>1) and (X>1) or not (B=0) and (A=2) or not (B=0) and (X>1)

(A>1) and (B=0) and not (A=2) and not (X/A>1)

条件覆盖

- 条件覆盖就是设计若干个测试用例，运行被测程序，使得程序中每个判断的每个条件的可能取值至少执行一次。
- 在图例中，我们事先可对所有条件的取值加以标记。例如，
- 对于第一个判断：
 - 条件 A＞1 取真为T_1，取假为$\overline{T_1}$
 条件 B＝0 取真为T_2，取假为$\overline{T_2}$

- 对于第二个判断：
 - 条件A＝2 取真为T_3，取假为 $\overline{T_3}$
 条件X＞1 取真为T_4，取假为 $\overline{T_4}$

测试用例	覆盖分支	条件取值
【(2, 0, 4)，(2, 0, 3)】	L1(c, e)	$T_1T_2T_3T_4$
【(1, 0, 1)，(1, 0, 1)】	L2(b, d)	$\overline{T_1}\overline{T_2}\overline{T_3}\overline{T_4}$
【(2, 1, 1)，(2, 1, 2)】	L3(b, e)	$T_1\overline{T_2}T_3\overline{T_4}$

或

测试用例	覆盖分支	条件取值
【(1, 0, 3)，(1, 0, 4)】	L3(b, e)	$\overline{T_1}T_2\overline{T_3}T_4$
【(2, 1, 1)，(2, 1, 2)】	L3(b, e)	$T_1T_2T_3T_4$

判定—条件覆盖

- 判定—条件覆盖就是设计足够的测试用例，使得判断中每个条件的所有可能取值至少执行一次，每个判断中的每个条件的可能取值至少执行一次。

测试用例	覆盖分支	条件取值
【(2, 0, 4)，(2, 0, 3)】	L1(c, e)	$T_1T_2T_3T_4$
【(1, 1, 1)，(1, 1, 1)】	L2(b, d)	$T_1T_2T_3T_4$

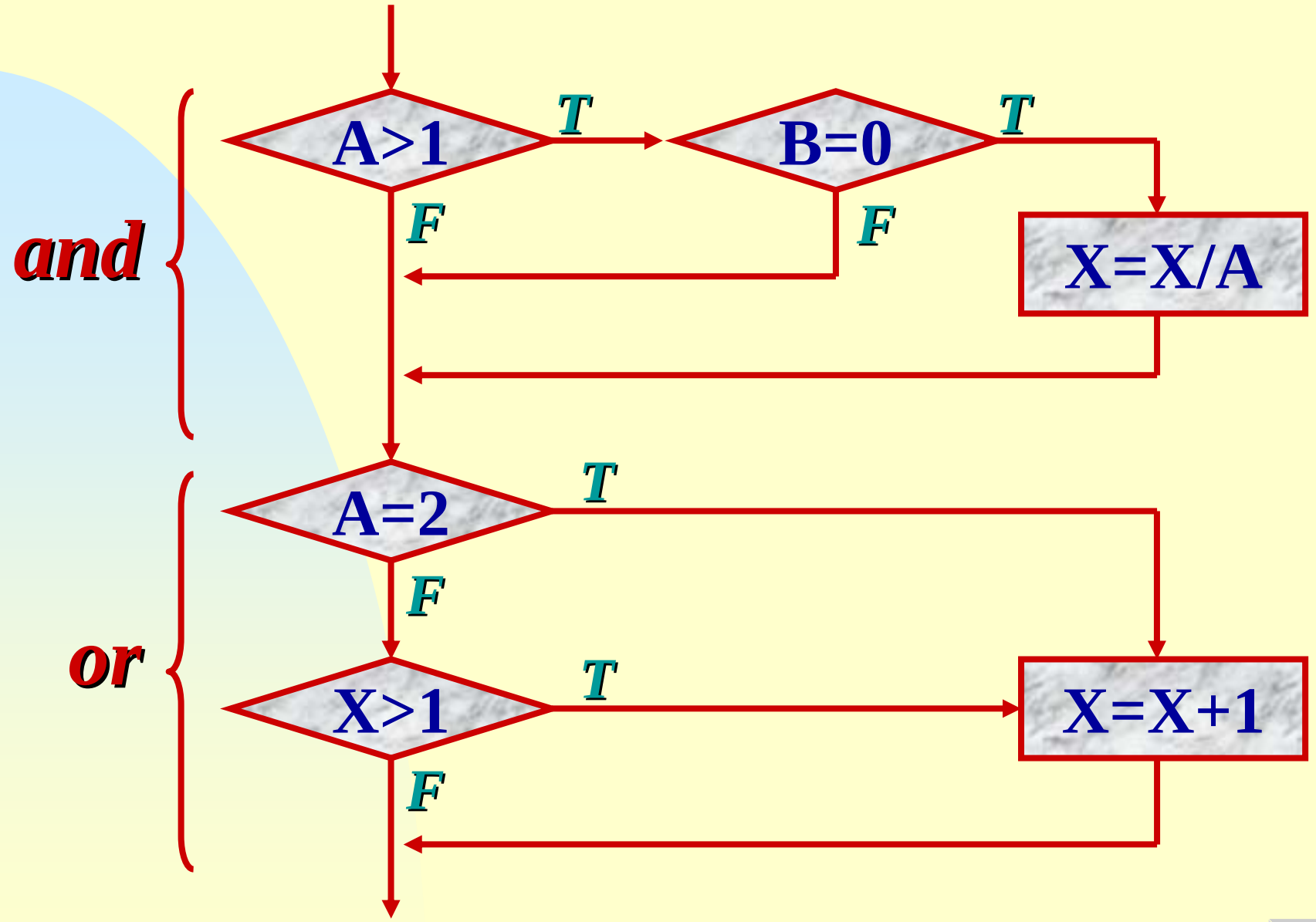
(A=2) and (B=0) or

(A>1) and (B=0) and (X/A>1)

not (A>1) and not (A=2) and not (X>1)

or

not (B=0) and not (A=2) and not (X>1)


and
A>1
T
B=0
T
F
F
X=X/A
or
A=2
T
F
X>1
T
X=X+1
F

条件组合覆盖

- 条件组合覆盖就是设计足够的测试用例，运行被测程序，使得每个判断的所有可能的条件取值组合至少执行一次。

 记 ① A＞1, B＝0 作 T_1T_2

 ② A＞1, B≠0 作 $T_1\overline{T_2}$

 ③ A≯1, B＝0 作 $\overline{T_1}T_2$

 ④ A≯1, B≠0 作 $\overline{T_1}\overline{T_2}$

⑤ A＝2, X＞1 作$T_3\overline{T_4}$

⑥ A＝2, X≯1 作$T_3\overline{T_4}$

⑦ A≠2, X＞1 作$\overline{T_3}T_4$

⑧ A≠2, X≯1 作$\overline{T_3}\overline{T_4}$

测试用例	覆盖条件	覆盖组合
【(2, 0, 4), (2, 0, 3)】	(L1)	$T_1\overline{T_2}T_3\overline{T_4}$ ①, ⑤
【(2, 1, 1), (2, 1, 2)】	(L3)	$T_1\overline{T_2}T_3\overline{T_4}$ ②, ⑥
【(1, 0, 3), (1, 0, 4)】	(L3)	$\overline{T_1}T_2\overline{T_3}T_4$ ③, ⑦
【(1, 1, 1), (1, 1, 1)】	(L2)	$\overline{T_1}\overline{T_2}\overline{T_3}\overline{T_4}$ ④, ⑧

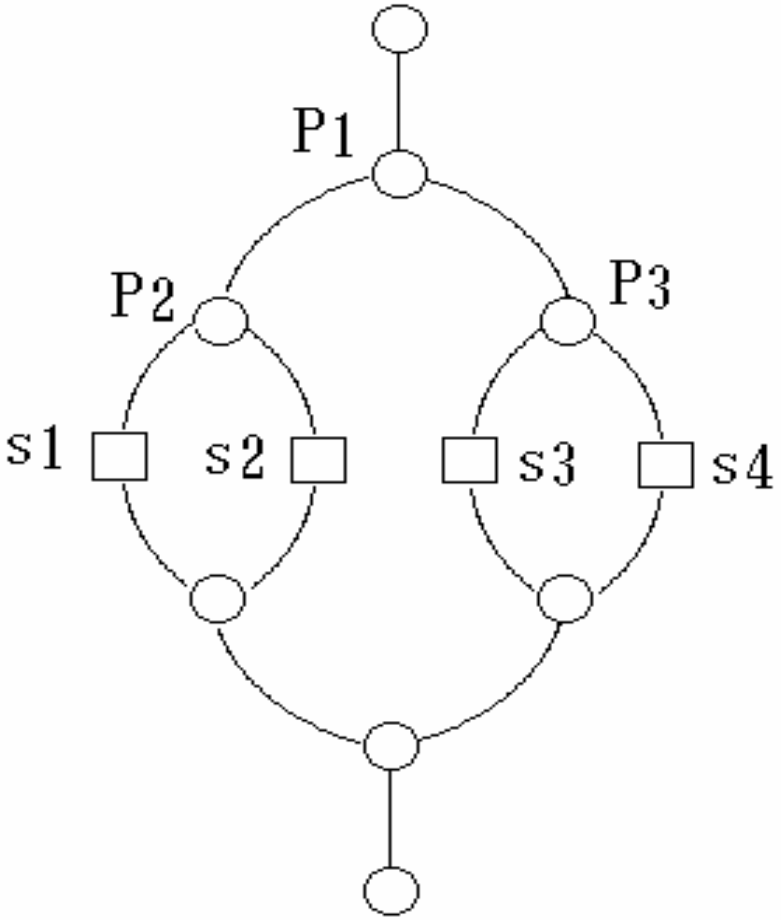
路径测试

- 路径测试就是设计足够的测试用例，覆盖程序中所有可能的路径。

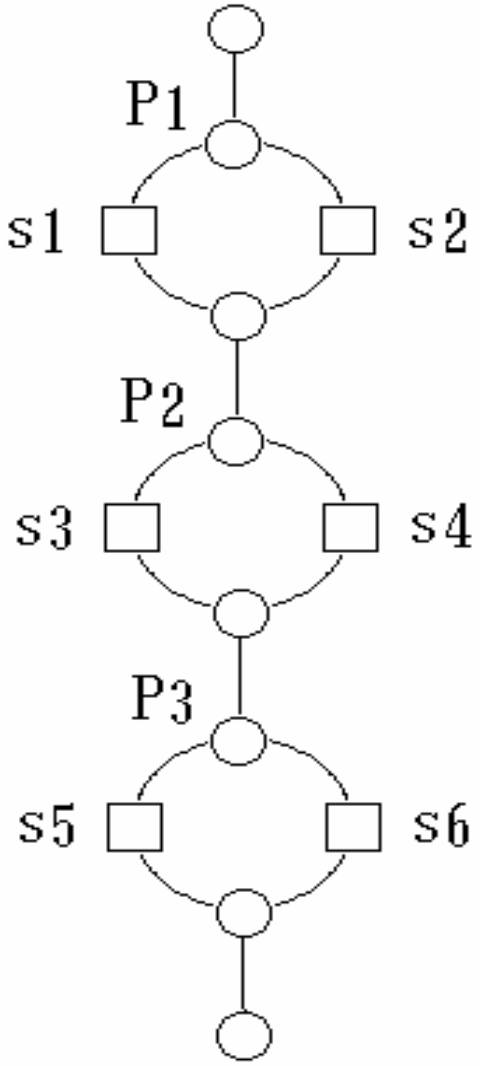
测 试 用 例	通过路径	覆盖条件
【(2, 0, 4), (2, 0, 3)】	ace (L1)	$T_1T_2T_3T_4$
【(1, 1, 1), (1, 1, 1)】	abd (L2)	$T_1T_2\overline{T_3}T_4$
【(1, 1, 2), (1, 1, 3)】	abe (L3)	$T_1T_2T_3\overline{T_4}$
【(3, 0, 3), (3, 0, 1)】	acd (L4)	$T_1\overline{T_2}T_3T_4$

条件测试路径选择

- 当程序中判定多于一个时，形成的分支结构可以分为两类：嵌套型分支结构和连锁型分支结构。
- 对于嵌套型分支结构，若有n个判定语句，需要$n+1$个测试用例；
- 对于连锁型分支结构， 若有n个判定语句，需要有2^n个测试用例，覆盖它的2^n条路径。



（a）嵌套型分支结构



（b）连锁型分支结构

循环测试路径选择

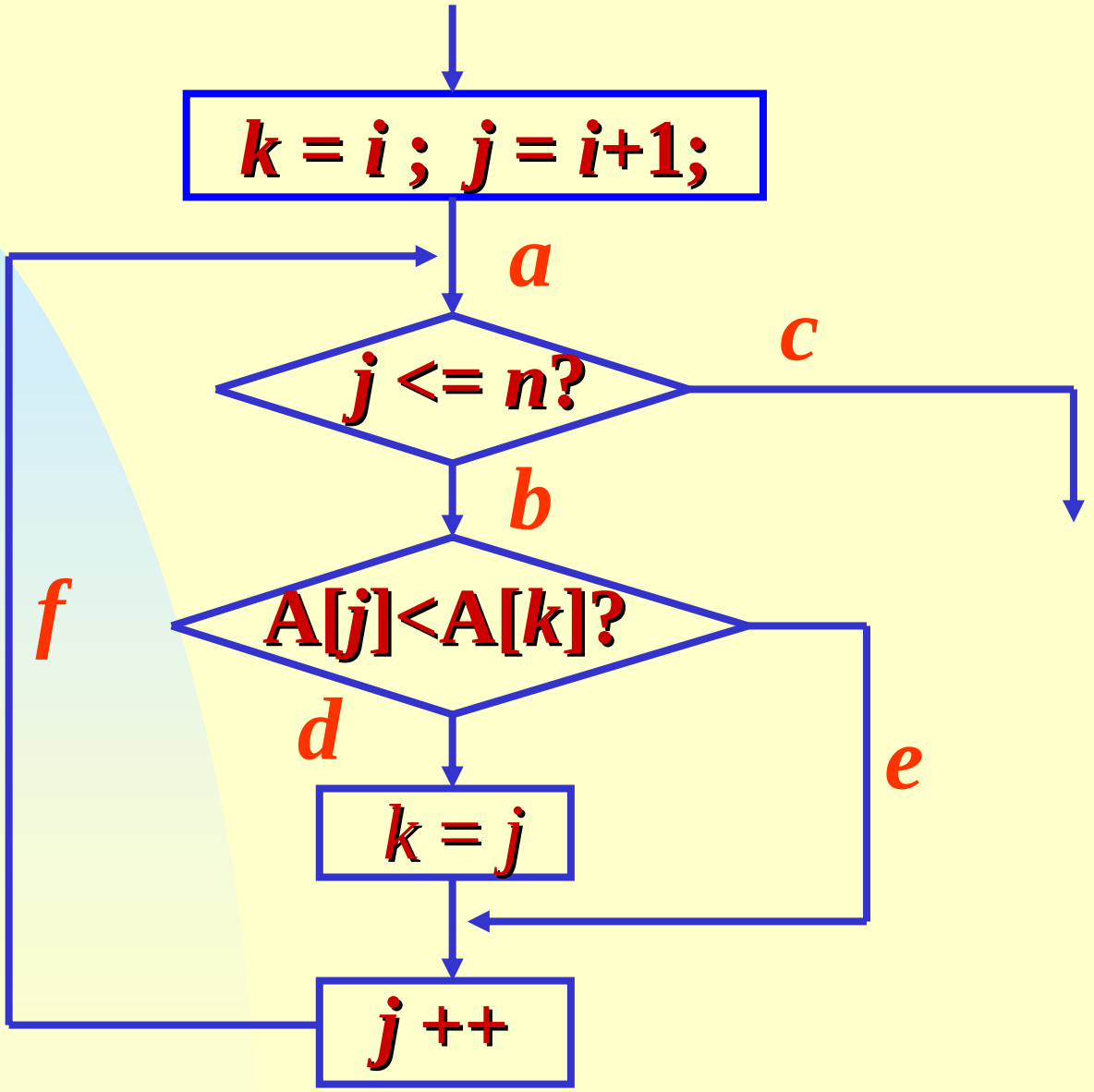
- 循环分为4种不同类型：简单循环、连锁循环、嵌套循环和非结构循环。

(1) 简单循环

① 零次循环：从循环入口到出口
② 一次循环：检查循环初始值
③ 二次循环：检查多次循环
④ *m*次循环： 检查在多次循环
⑤ 最大次数循环、比最大次数多一次、少一次的循环。

例：求最小值

```
k = i;
for ( j = i+1;  j <= n;  j++ )
    if ( A[j] < A[k] )  k = j;
```


k = i ; j = i+1;
a
j <= n?
c
b
A[j]<A[k]?
f
d
e
k = j
j ++

测试用例选择

循环	*i*	*n*	A[*i*]	A[*i*+1]	A[*i*+2]	*k*	路 径
0	1	1				*i*	*a c*
1	1	2	1	2		*i*	*a b e f c*
			2	1		*i*+1	*a b d f c*
2	1	3	1	2	3	*i*	*a b e f e f c*
			2	3	1	*i*+2	*a b e f d f c*
			3	2	1	*i*+2	*a b d f d f c*
			3	1	2	*i*+1	*a b d f e f c*

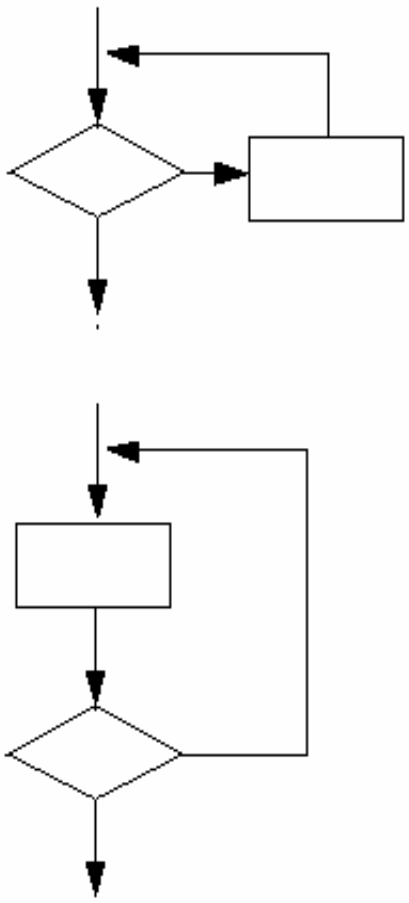
d 改 *k* 的值，*e* 不改 *k* 的值

(2) 嵌套循环

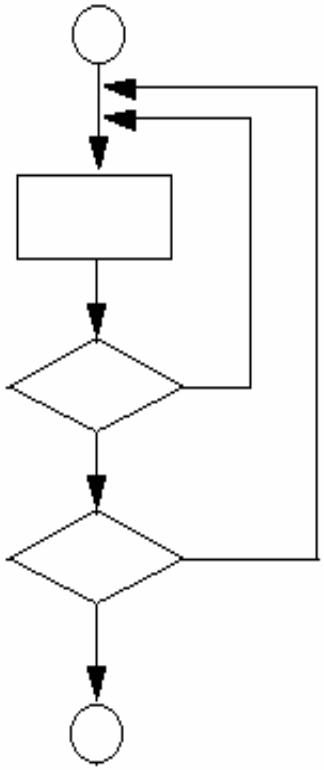
① 对最内层循环做简单循环的全部测试。所有其它层的循环变量置为最小值；

② 逐步外推，对其外面一层循环进行测试。测试时保持所有外层循环的循环变量取最小值，所有其它嵌套内层循环的循环变量取“典型”值。。

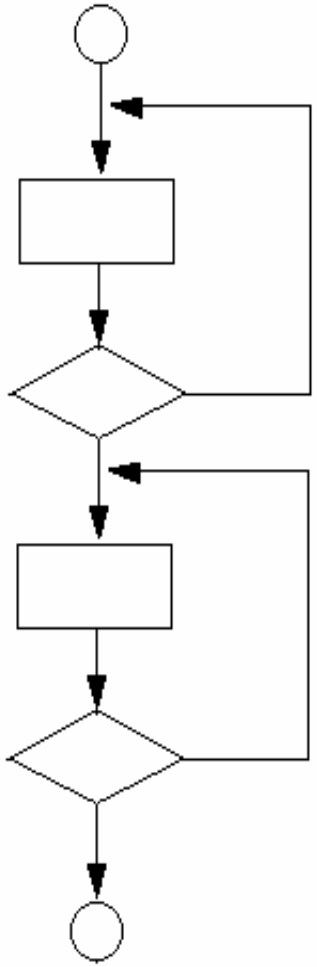
③ 反复进行，直到所有各层循环测试完毕。



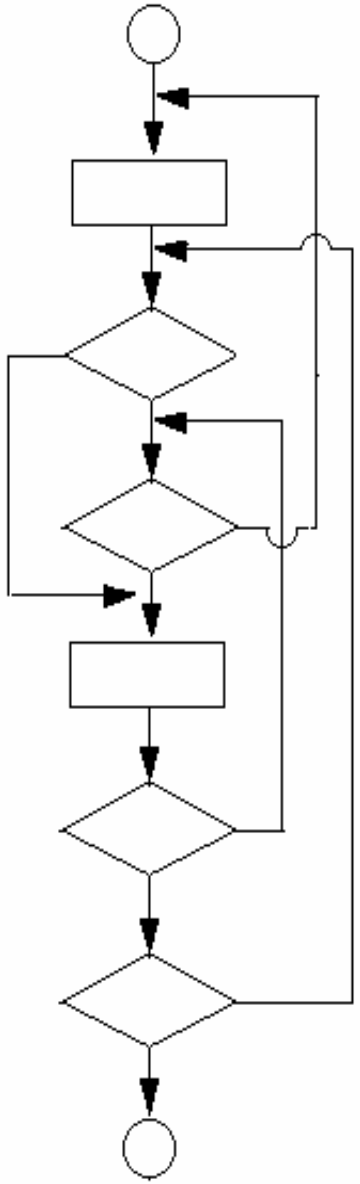
简单循环



嵌套循环



连锁循环



非结构循环

④ 对全部各层循环同时取最小循环次数，或者同时取最大循环次数

(3) 连锁循环

如果各个循环互相独立，则可以用与简单循环相同的方法进行测试。但如果几个循环不是互相独立的，则需要使用测试嵌套循环的办法来处理。

(4) 非结构循环

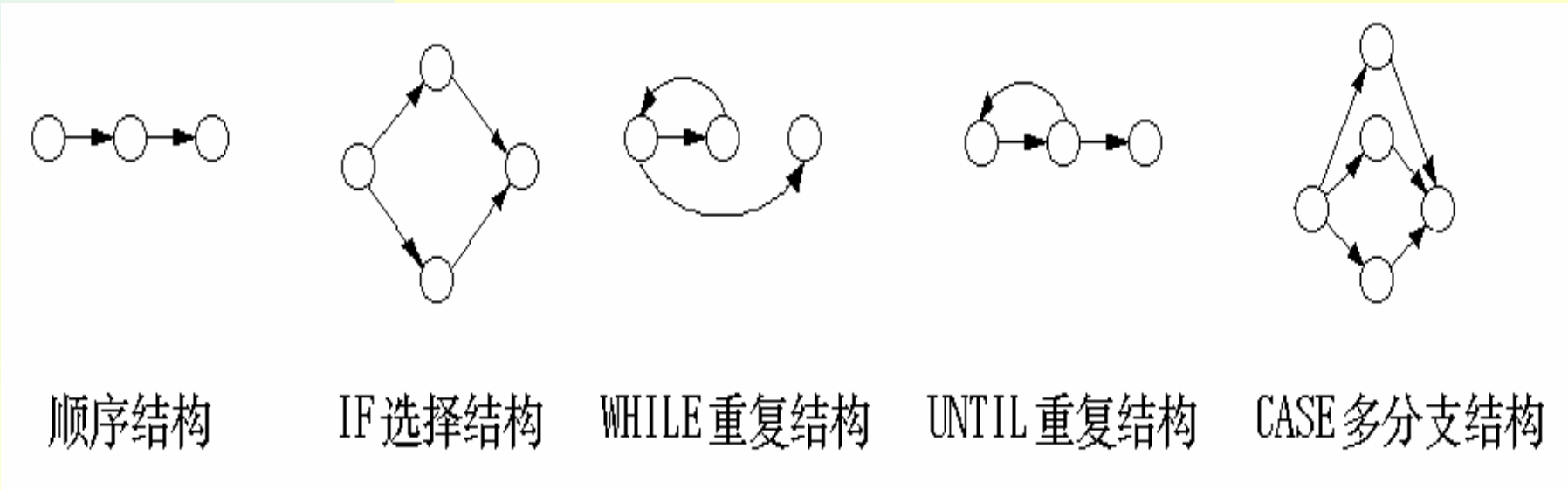
这一类循环应该使用结构化程序设计方法重新设计测试用例。

基本路径测试

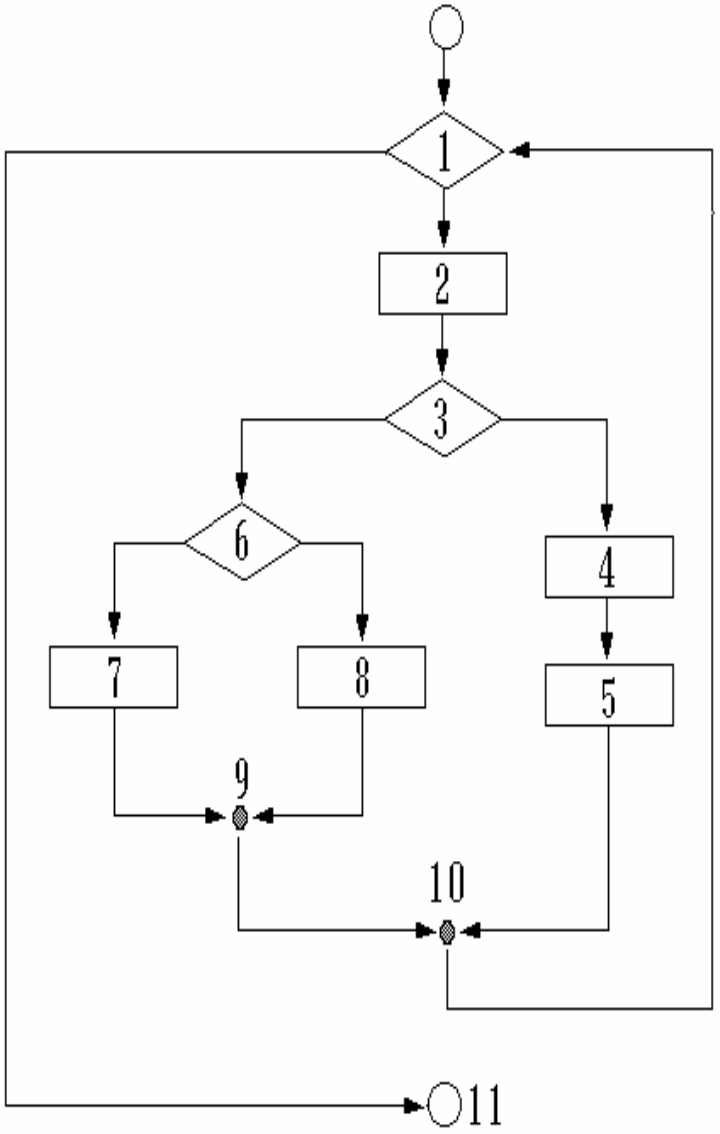
- 基本路径测试方法把覆盖的路径数压缩到一定限度内，程序中的循环体最多只执行一次。
- 它是在程序控制流图的基础上，分析控制构造的环路复杂性，导出基本可执行路径集合，设计测试用例的方法。设计出的测试用例要保证在测试中，程序的每一个可执行语句至少要执行一次。

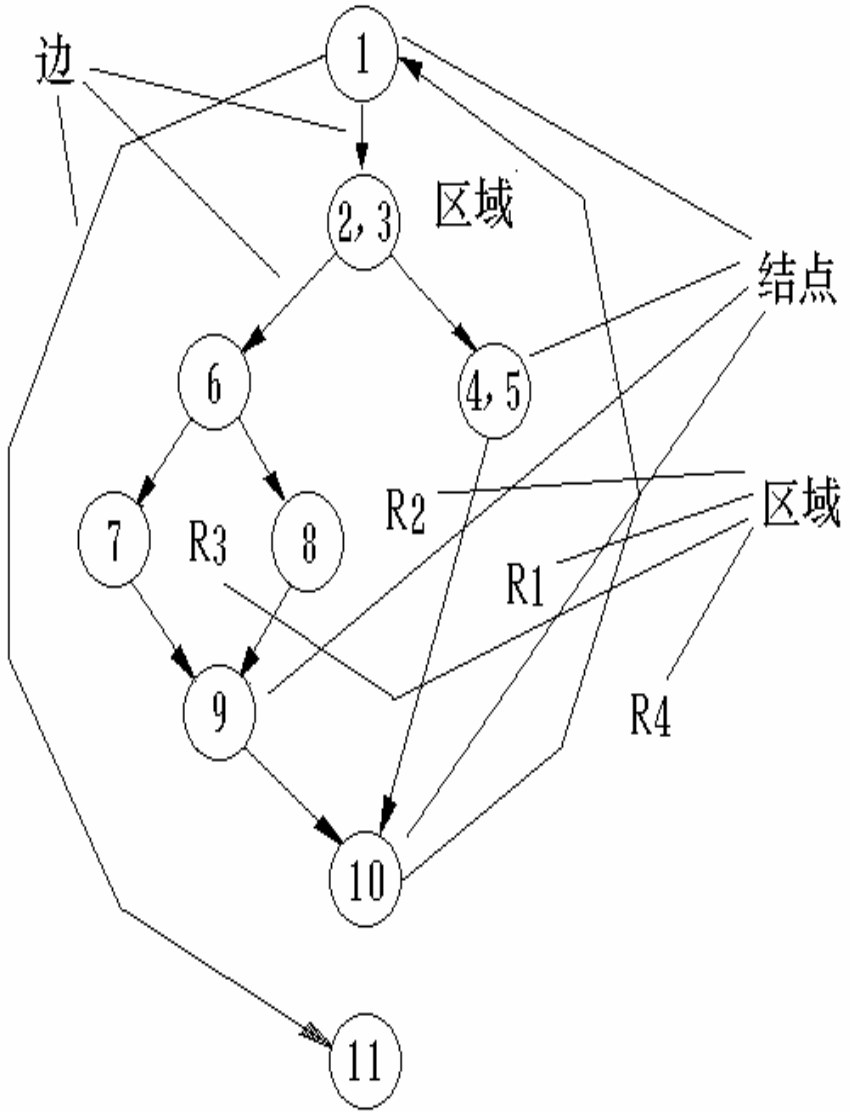
1. 程序的控制流图

- 符号○为控制流图的一个结点，表示一个或多个无分支的PDL语句或源程序语句。箭头为边，表示控制流的方向。



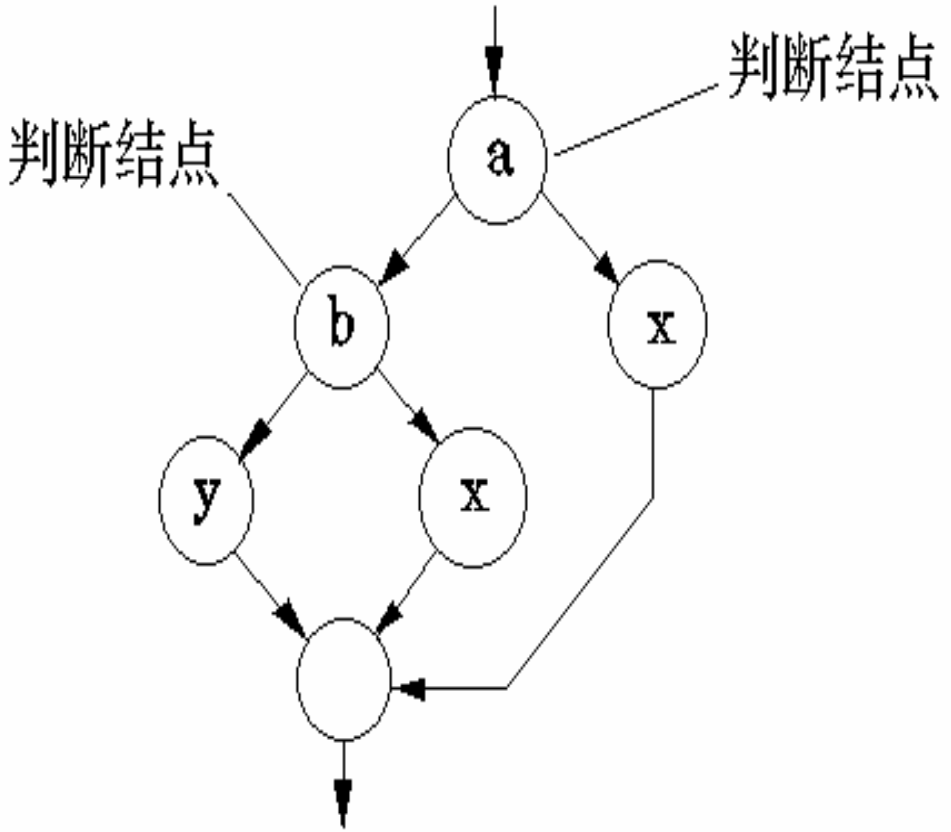
- 在选择或多分支结构中，分支的汇聚处应有一个汇聚结点。
- 边和结点圈定的区域叫做区域，当对区域计数时，图形外的区域也应记为一个区域。
- 如果判断中的条件表达式是由一个或多个逻辑运算符 (OR，AND, ...) 连接的复合条件表达式，则需改为一系列只有单个条件的嵌套的判断。


1
2
3
6
4
7
8
5
9
10
11


边
1
2, 3
区域
结点
6
4, 5
7
R3
8
R2
区域
R1
9
R4
10
11

```
      .
      .
      .
if   a OR b
  then  procedure  x
  else  procedure  y;
      .
      .
      .
```

(a)



(b)

2. 程序环路复杂性

- 程序的环路复杂性给出了**程序基本路径集中的独立路径条数**，这是确保程序中每个可执行语句至少执行一次所必需的测试用例数目的上界。
- 从控制流图来看，一条独立路径是至少包含有一条在其它独立路径中从未有过的边的路径。

- 例如，在图示的控制流图中，一组独立的路径是
 path1：1 - 11
 path2：1 - 2 - 3 - 4 - 5 - 10 - 1 - 11
 path3：1 - 2 - 3 - 6 - 8 - 9 - 10 - 1 - 11
 path4：1 - 2 - 3 - 6 - 7 - 9 - 10 - 1 - 11
- 路径 path1，path2，path3，path4 组成了控制流图的一个基本路径集。

3. 导出测试用例

- 导出测试用例，确保基本路径集中的每一条路径的执行。
- 根据判断结点给出的条件，选择适当的数据以保证某一条路径可以被测试到 — 用逻辑覆盖方法。

- 每个测试用例执行之后，与预期结果进行比较。如果所有测试用例都执行完毕，则可以确信程序中所有的可执行语句至少被执行了一次。
- 必须注意，一些独立的路径(如例中的路径1)，往往不是完全孤立的，有时它是程序正常的控制流的一部分，这时，这些路径的测试可以是另一条路径测试的一部分。

黑盒测试的测试用例设计

- 等价类划分
- 边界值分析
- 错误推测法
- 因果图

等价类划分

- 等价类划分是一种典型的黑盒测试方法，使用这一方法时，完全不考虑程序的内部结构，只依据程序的规格说明来设计测试用例。
- 等价类划分方法把所有可能的输入数据，即程序的输入域划分成若干部分，然后从每一部分中选取少数有代表性的数据做为测试用例。

- 使用这一方法设计测试用例要经历划分等价类（列出等价类表）和选取测试用例两步。
- 划分等价类
 等价类是指某个输入域的子集合。在该子集合中，各个输入数据对于揭露程序中的错误都是等效的。测试某等价类的代表值就等价于对这一类其它值的测试。

- 等价类的划分有两种不同的情况：
 ① **有效等价类**：是指对于程序的规格说明来说，是合理的，有意义的输入数据构成的集合。
 ② **无效等价类**：是指对于程序的规格说明来说，是不合理的，无意义的输入数据构成的集合。
- 在设计测试用例时，要同时考虑有效等价类和无效等价类的设计

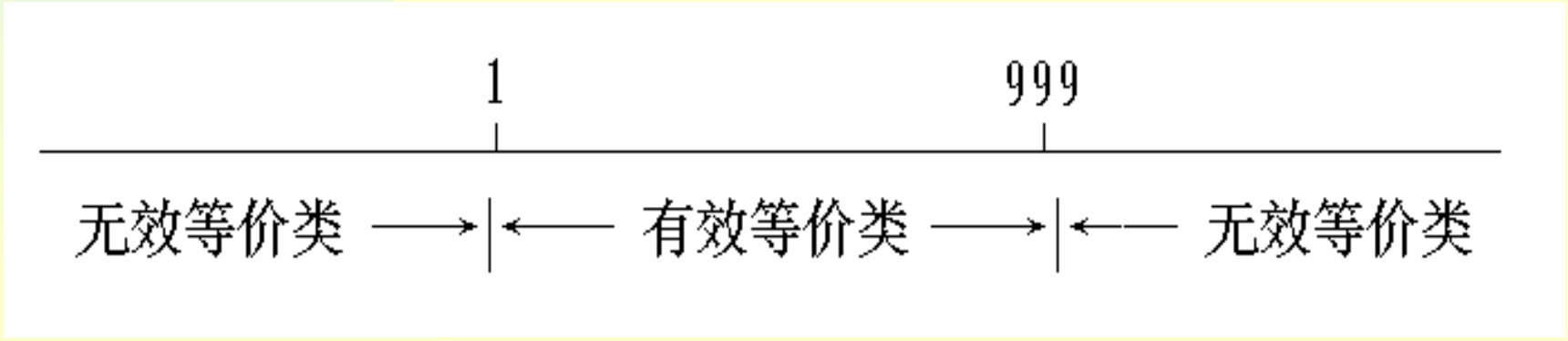
- 划分等价类等价类的原则。
 (1) 如果输入条件规定了取值范围，或值的个数，则可以确立一个有效等价类和两个无效等价类。

- 例如，在程序的规格说明中，对输入条件有一句话：

 “…… 项数可以从1到999 ……”

 则有效等价类是“1≤项数≤999”

 两个无效等价类是“项数＜1”或“项数＞999”。在数轴上表示成：



(2) 如果输入条件规定了输入值的集合，或者是规定了“必须如何”的条件，这时可确立一个有效等价类和一个无效等价类。

- 例如，在Pascal语言中对变量标识符规定为“以字母打头的……串”。那么所有以字母打头的构成有效等价类，而不在此集合内（不以字母打头）的归于无效等价类。

(3) 如果输入条件是一个布尔量，则可以确定一个有效等价类和一个无效等价类。

(4) 如果规定了输入数据的一组值，而且程序要对每个输入值分别进行处理。这时可为 每一个输入值确立一个有效等价类，此外针对这组值确立一个无效等价类，它是所有不允许的输入值的集合。

- 例如，在教师上岗方案中规定对教授、副教授、讲师和助教分别计算分数，做相应的处理。因此可以确定4个有效等价类为教授、副教授、讲师和助教，一个无效等价类，它是所有不符合以上身分的人员的输入值的集合。

(5) 如果规定了输入数据必须遵守的规则，则可以确立一个有效等价类（符合规则）和若干个无效等价类（从不同角度违反规则）。

- 例如，Pascal语言规定“一个语句必须以分号‘;’结束”。这时，可以确定一个有效等价类“以‘;’结束”，若干个无效等价类“以‘:’结束”、“以‘,’结束”、“以‘ ’结束”、“以LF结束”等。
- 确立测试用例

输入条件	有效等价类	无效等价类
……	……	……
……	……	……

- 再从划分出的等价类中按以下原则选择测试用例：
 (1) 为每一个等价类规定一个唯一编号；
 (2) 设计一个新的测试用例，使其尽可能多地覆盖尚未被覆盖的有效等价类，重复这一步，直到所有的有效等价类都被覆盖为止；
 (3) 设计一个新的测试用例，使其仅覆盖一个尚未被覆盖的无效等价类，重复这一步，直到所有的无效等价类都被覆盖为止。

- 用等价类划分法设计测试用例的实例

 在某一PASCAL语言版本中规定：“标识符是由字母开头，后跟字母或数字的任意组合构成。有效字符数为8个，最大字符数为80个。”

 并且规定：“标识符必须先说明，再使用。”“在同一说明语句中，标识符至少必须有一个。”

用等价类划分方法，建立输入等价类表：

输入条件	有效等价类	无效等价类
标识符个数	1个（1），多个（2）	0个（3）
标识符字符数	1~8个（4）	0个（5），>8个（6），>80个（7）
标识符组成	字母（8），数字（9）	非字母数字字符（10），保留字（11）
第一个字符	字母（12）	非字母（13）
标识符使用	先说明后使用（14）	未说明已使用（15）

- 下面选取了9个测试用例，它们覆盖了所有的等价类。

① VAR *x*，T1234567：REAL；
BEGIN *x* := 3.414；
T1234567 := 2.732；
……
(1), (2), (4), (8), (9), (12), (14)

② VAR ：REAL； (3)

③ VAR *x*， ：REAL; (5)

④ VAR T12345678：REAL； (6)

⑤ VAR T12345.....：REAL； (7)

多于80个字符

⑥ VAR T$：CHAR； (10)

⑦ VAR GOTO：INTEGER； (11)

⑧ VAR 2T：REAL； (13)

⑨ VAR PAR：REAL； (15)

BEGIN

PAP := SIN (3.14 * 0.8) / 6；

边界值分析

- 边界值分析也是一种黑盒测试方法，是对等价类划分方法的补充。
- 人们从长期的测试工作经验得知，**大量的错误是发生在输入或输出范围的边界上，而不是在输入范围的内部。**因此针对各种边界情况设计测试用例，可以查出更多的错误。

- 比如，在做三角形计算时，要输入三角形的三个边长：A、B和C。 我们应注意到这三个数值应当满足

 A＞0、B＞0、C＞0、

 A＋B＞C、A＋C＞B、B＋C＞A，才能构成三角形。但如果把六个不等式中的任何一个大于号“＞”错写成大于等于号“≥”，那就不能构成三角形。问题恰出现在容易被疏忽的边界附近。

- 这里所说的边界是指，相当于输入等价类和输出等价类而言，稍高于其边界值及稍低于其边界值的一些特定情况。
- 使用边界值分析方法设计测试用例，首先应确定边界情况。应当选取正好等于，刚刚大于，或刚刚小于边界的值做为测试数据，而不是选取等价类中的典型值或任意值做为测试数据。



错误推测法

- 人们也可以靠经验和直觉推测程序中可能存在的各种错误，从而有针对性地编写检查这些错误的例子。这就是错误推测法。
- 错误推测法的基本想法是：列举出程序中所有可能有的错误和容易发生错误的特殊情况，根据它们选择测试用例。



因果图

- 因果图的适用范围
 如果在测试时必须考虑输入条件的各种组合，可使用一种适合于描述对于多种条件的组合，相应产生多个动作的形式来设计测试用例，这就需要利用因果图。
 因果图方法最终生成的就是判定表。它适合于检查程序输入条件的各种组合情况。

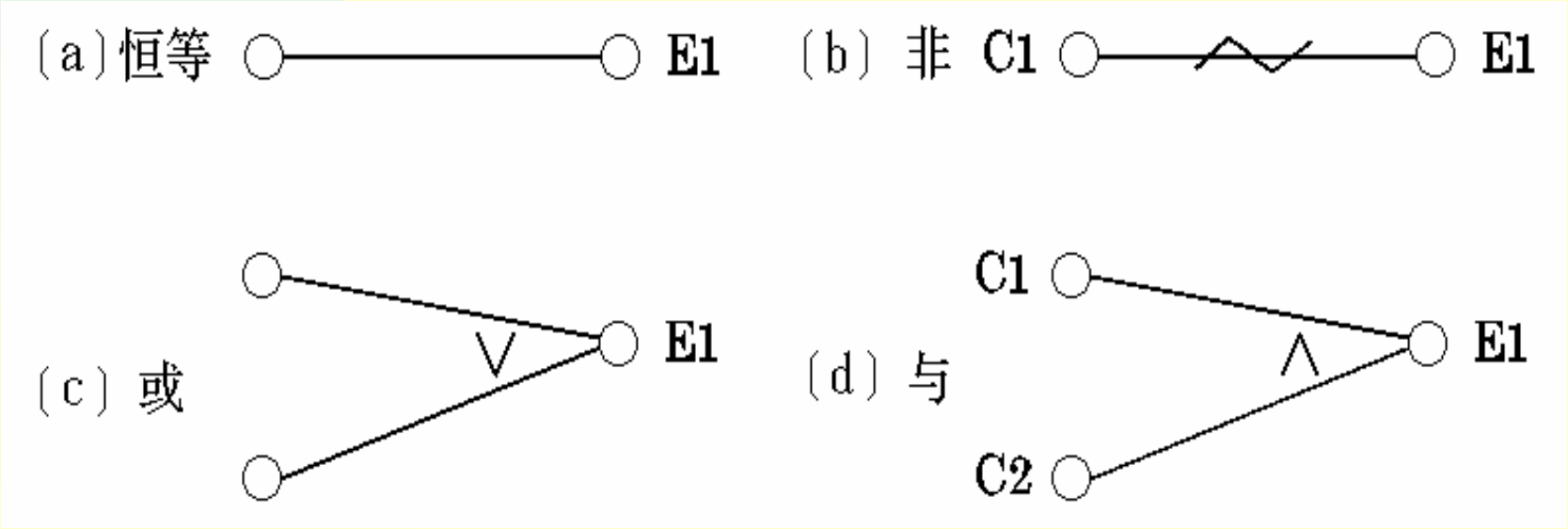
- 用因果图生成测试用例的基本步骤
 (1) 分析软件规格说明描述中，哪些是原因 (即输入条件或输入条件的等价类)，哪些是结果 (即输出条件)，并给每个原因和结果赋予一个标识符。
 (2) 分析软件规格说明描述中的语义，找出原因与结果之间，原因与原因之间对应的是什么关系? 根据这些关系，画出因果图。

(3) 由于语法或环境限制，有些原因与原因之间，原因与结果之间的组合情况不可能出现。为表明这些特殊情况，在因果图上用一些记号标明约束或限制条件。

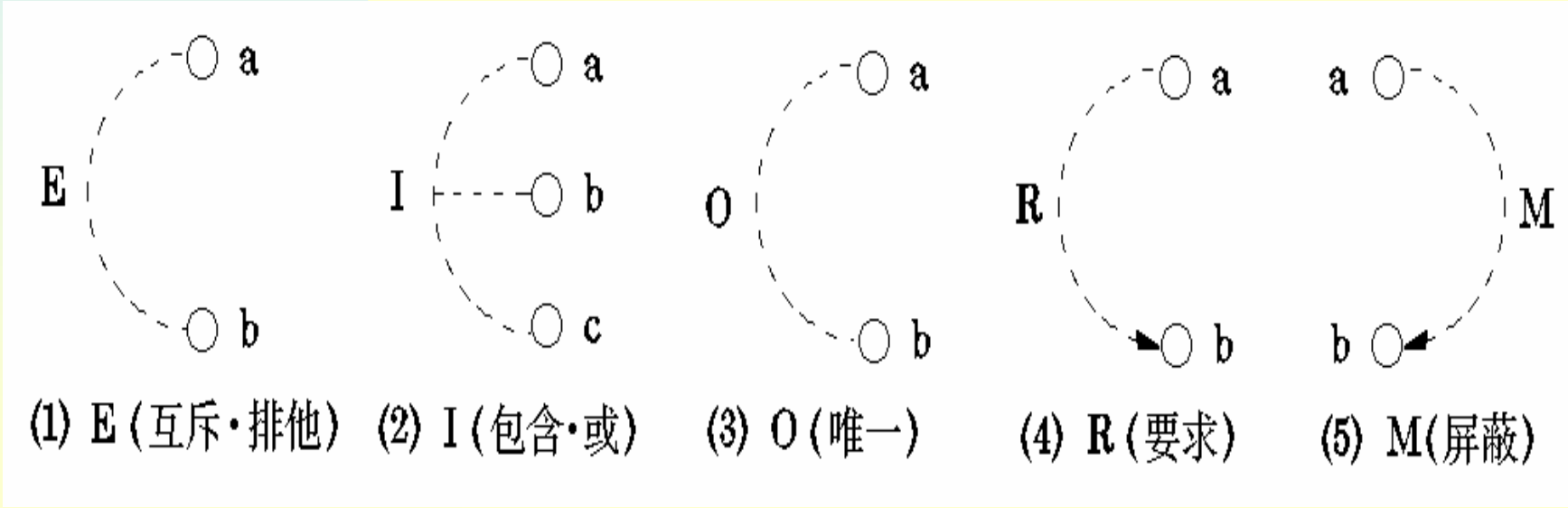
(4) 把因果图转换成判定表。

(5) 把判定表的每一列拿出来作为依据，设计测试用例。

- 在因果图中出现的基本符号
 通常在因果图中用Ci表示原因，用Ei表示结果，各结点表示状态，可取值“0”或“1”。“0”表示某状态不出现，“1”表示某状态出现。
- 主要的原因和结果之间的关系有：



- 表示约束条件的符号
 为了表示原因与原因之间，结果与结果之间可能存在的约束条件，在因果图中可以附加一些表示约束条件的符号。



- 例如，有一个处理单价为5角钱的饮料的自动售货机软件测试用例的设计。其规格说明如下：
若投入5角钱或1元钱的硬币，押下〖橙汁〗或〖啤酒〗的按钮，则相应的饮料就送出来。若售货机没有零钱找，则一个显示〖零钱找完〗的红灯亮，这时在投入1元硬币并押下按钮后，饮料不送出来而且1元硬币也退出来；若有零钱找，则显示〖零钱找完〗的红灯灭，在送出饮料的同时退还5角硬币。”

(1) 分析这一段说明，列出原因和结果

原因: 1. 售货机有零钱找

2. 投入1元硬币

3. 投入5角硬币

4. 押下橙汁按钮

5. 押下啤酒按钮

建立中间结点，表示处理中间状态

11. 投入1元硬币且押下饮料按钮

12. 押下〖橙汁〗或〖啤酒〗的按钮

13. 应当找5角零钱并且售货机有零钱找

14. 钱已付清

结果： 21. 售货机〖零钱找完〗灯亮

22. 退还1元硬币

23. 退还5角硬币

24. 送出橙汁饮料

25. 送出啤酒饮料

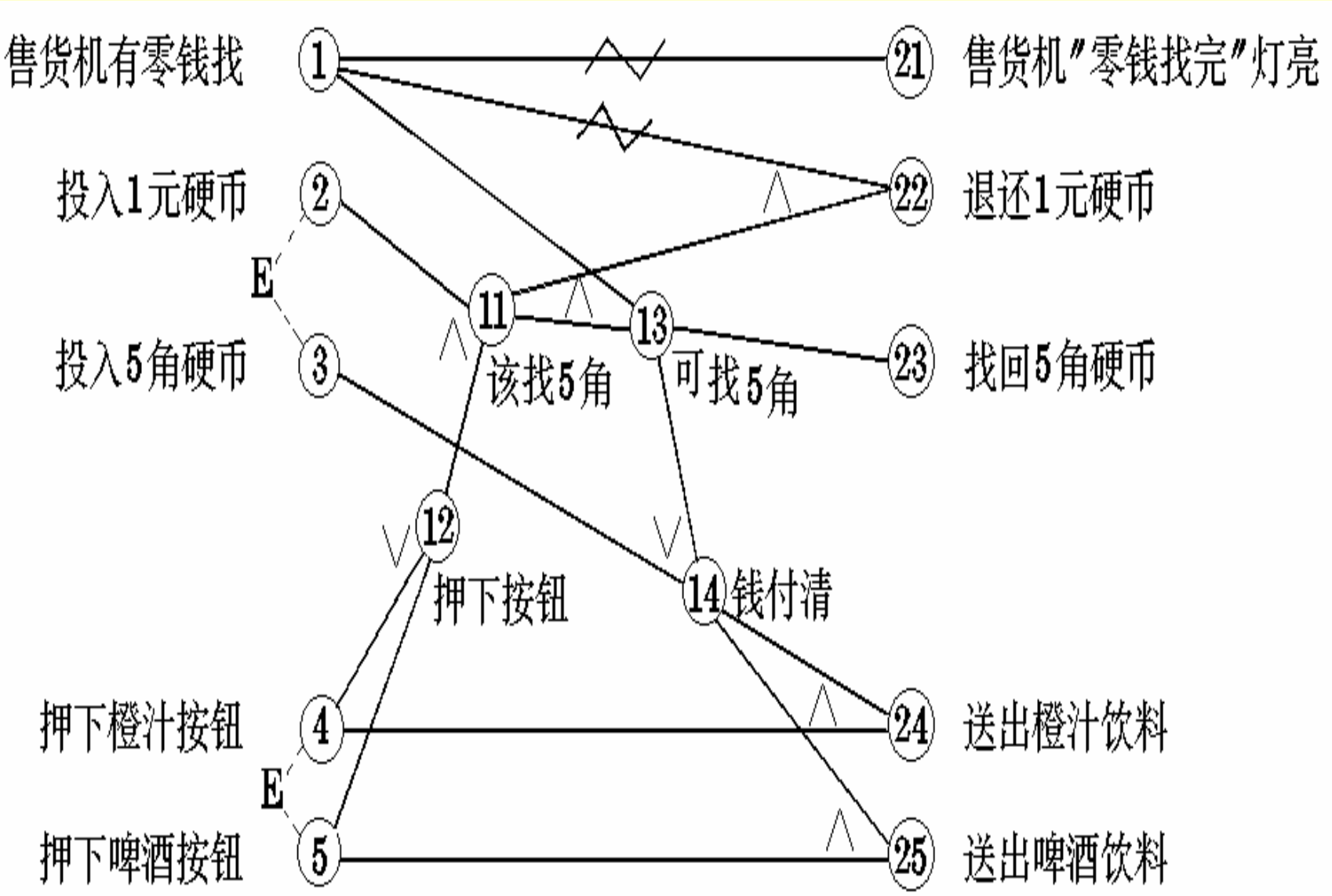
(2) 画出因果图。所有原因结点列在左边，所有结果结点列在右边。

(3) 由于 2 与 3 ，4 与 5 不能同时发生，

分别加上约束条件E。

(4) 因果图

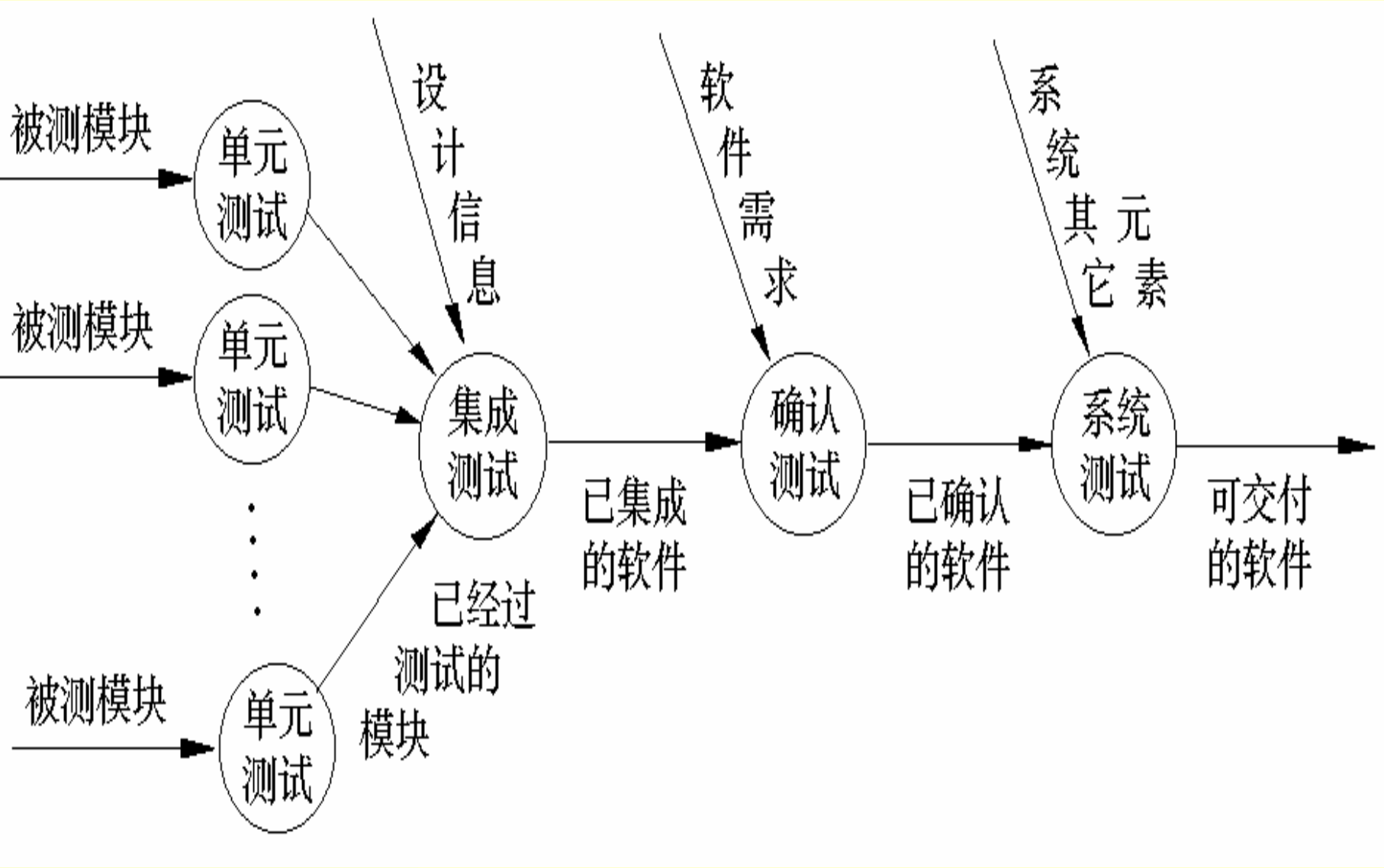
(5) 转换成判定表


售货机有零钱找
1
21
售货机"零钱找完"灯亮
投入1元硬币
2
22
退还1元硬币
E
11
13
投入5角硬币
3
该找5角
可找5角
23
找回5角硬币
12
押下按钮
14
钱付清
押下橙汁按钮
4
24
送出橙汁饮料
E
押下啤酒按钮
5
25
送出啤酒饮料

序号		1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1	2
条件	①	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	②	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
	③	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0
	④	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
	⑤	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
中间结果	⑪						1	1	0		0	0	0		0	0	0						1	1	0		0	0	0		0	0	0
	⑫						1	1	0		1	1	0		1	1	0						1	1	0		1	1	0		1	1	0
	⑬						1	1	0		0	0	0		0	0	0						0	0	0		0	0	0		0	0	0
	⑭						1	1	0		1	1	1		0	0	0						0	0	0		1	1	1		0	0	0
结果	㉑						0	0	0		0	0	0		0	0	0						1	1	1		1	1	1		1	1	1
	㉒						0	0	0		0	0	0		0	0	0						1	1	0		0	0	0		0	0	0
	㉓						1	1	0		0	0	0		0	0	0						0	0	0		0	0	0		0	0	0
	㉔						1	0	0		1	0	0		0	0	0						0	0	0		1	0	0		0	0	0
	㉕						0	1	0		0	1	0		0	0	0						0	0	0		0	1	0		0	0	0
测试用例							Y	Y	Y		Y	Y	Y		Y	Y							Y	Y	Y		Y	Y	Y		Y	Y	

软件测试的策略

- 测试过程按4个步骤进行，即单元测试、组装测试、确认测试和系统测试。
- 开始是单元测试，集中对用源代码实现的每一个程序单元进行测试，检查各个程序模块是否正确地实现了规定的功能。


被测模块
单元测试
被测模块
单元测试
被测模块
单元测试
设计信息
集成测试
已经过测试的模块
已集成的软件
软件需求
确认测试
已确认的软件
系统其它元素
系统测试
可交付的软件

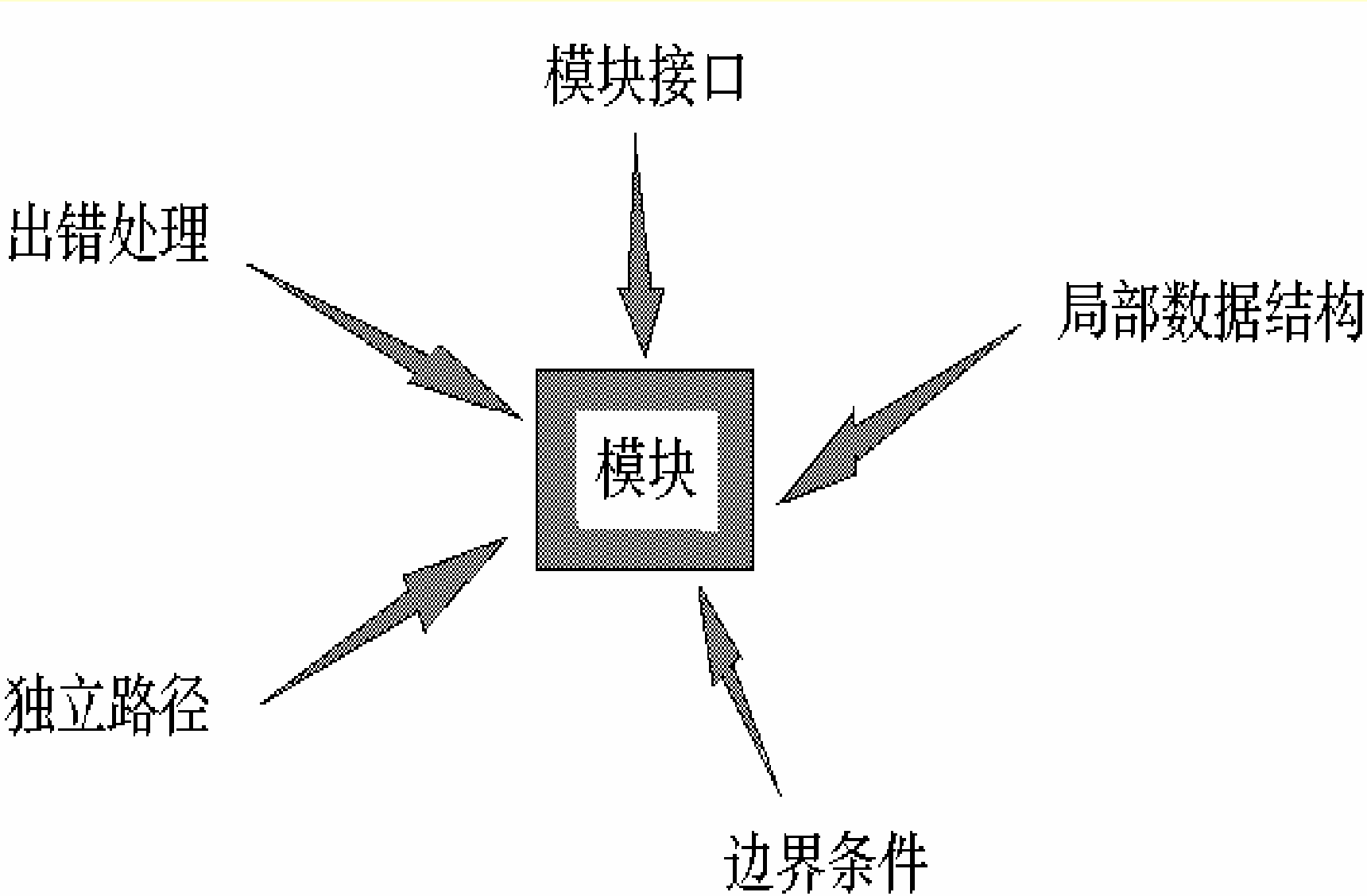
- **组装测试**把已测试过的模块组装起来，主要对与设计相关的软件体系结构的构造进行测试。
- **确认测试**则是要检查已实现的软件是否满足了需求规格说明中确定了的各种需求，以及软件配置是否完全、正确。
- **系统测试**把已经经过确认的软件纳入实际运行环境中，与其它系统成份组合在一起进行测试。

单元测试 (Unit Testing)

- 单元测试又称模块测试，是针对软件设计的最小单位 — 程序模块，进行正确性检验的测试工作。其目的在于发现各模块内部可能存在的各种差错。
- 单元测试需要从程序的内部结构出发设计测试用例。多个模块可以平行地独立进行单元测试。

1. 单元测试的内容

- 在单元测试时，测试者需要依据详细设计说明书和源程序清单，了解该模块的I/O条件和模块的逻辑结构，主要采用白盒测试的测试用例，辅之以黑盒测试的测试用例，使之对任何合理的输入和不合理的输入，都能鉴别和响应。


模块接口
出错处理
局部数据结构
模块
独立路径
边界条件

(1) 模块接口测试

- 在单元测试的开始，应对通过被测模块的数据流进行测试。测试项目包括：
 - 调用本模块的输入参数是否正确；
 - 本模块调用子模块时输入给子模块的参数是否正确；
 - 全局量的定义在各模块中是否一致；

- 在做内外存交换时要考虑：
 - 文件属性是否正确；
 - OPEN与CLOSE语句是否正确；
 - 缓冲区容量与记录长度是否匹配；
 - 在进行读写操作之前是否打开了文件；
 - 在结束文件处理时是否关闭了文件；
 - 正文书写／输入错误，
 - I／O错误是否检查并做了处理。

(2) 局部数据结构测试

- 不正确或不一致的数据类型说明
- 使用尚未赋值或尚未初始化的变量
- 错误的初始值或错误的缺省值
- 变量名拼写错或书写错
- 不一致的数据类型
- 全局数据对模块的影响

(3) 路径测试

- 选择适当的测试用例，对模块中重要的执行路径进行测试。
- 应当设计测试用例查找由于错误的计算、不正确的比较或不正常的控制流而导致的错误。
- 对基本执行路径和循环进行测试可以发现大量的路径错误。

(4) 错误处理测试

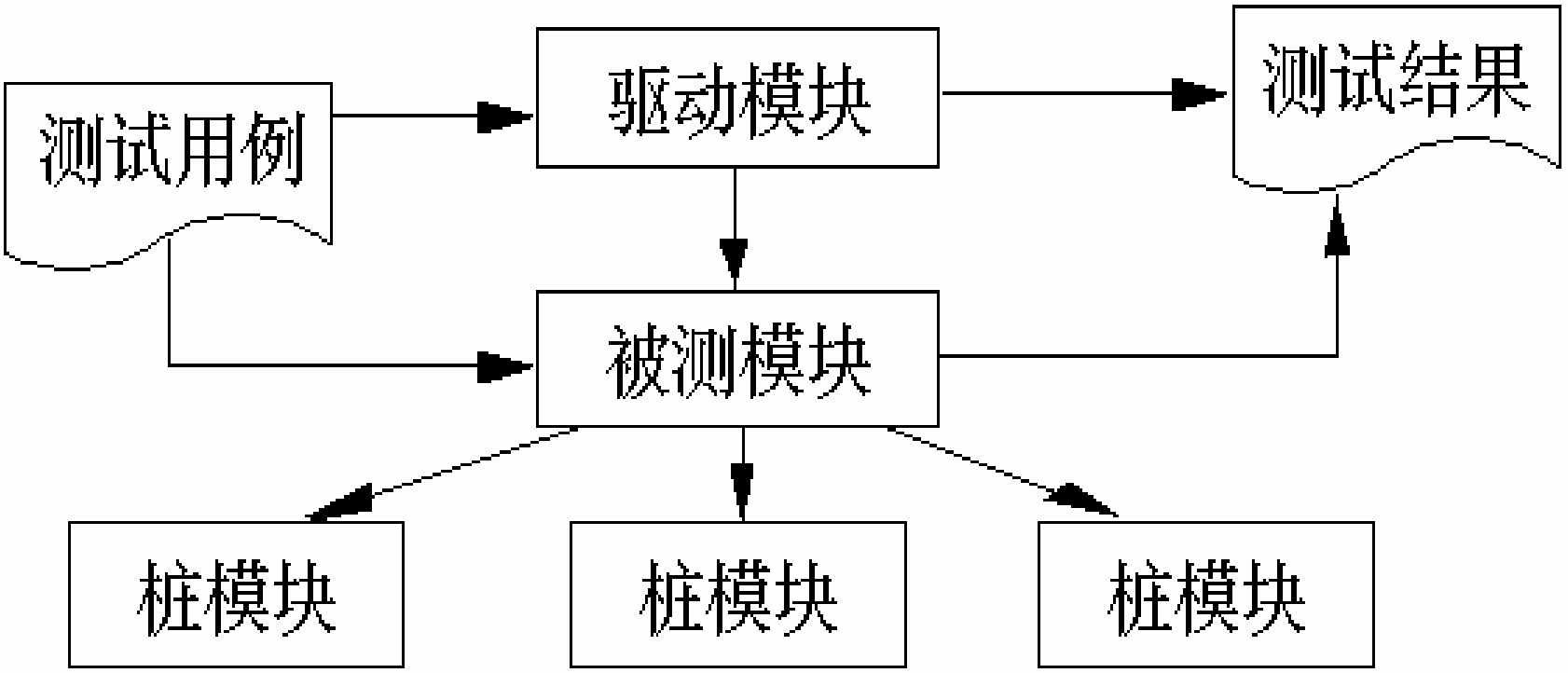
- 出错的描述是否难以理解
- 出错的描述是否能够对错误定位
- 显示的错误与实际的错误是否相符
- 对错误条件的处理正确与否
- 在对错误进行处理之前，错误条件是否已经引起系统的干预等

(5) 边界测试

- 注意数据流、控制流中刚好等于、大于或小于确定的比较值时出错的可能性。对这些地方要仔细地选择测试用例，认真加以测试。
- 如果对模块运行时间有要求的话，还要专门进行关键路径测试，以确定最坏情况下和平均意义下影响模块运行时间的因素。

2. 单元测试的步骤

- 模块并不是一个独立的程序，在考虑测试模块时，同时要考虑它和外界的联系，用一些辅助模块去模拟与被测模块相联系的其它模块。
 - 驱动模块 (driver)
 - 桩模块 (stub) —— 存根模块


测试用例
驱动模块
测试结果
被测模块
桩模块
桩模块
桩模块

- 如果一个模块要完成多种功能，可以将这个模块看成由几个小程序组成。必须对其中的每个小程序先进行单元测试要做的工作，对关键模块还要做性能测试。
- 对支持某些标准规程的程序，更要着手进行互联测试。有人把这种情况特别称为模块测试，以区别单元测试。

组装测试（Integrated Testing）

- 组装测试 (集成测试、联合测试)
- 通常，在单元测试的基础上，需要将所有模块按照设计要求组装成为系统。这时需要考虑的问题是：
 - 在把各个模块连接起来的时候，穿越模块接口的数据是否会丢失；
 - 一个模块的功能是否会对另一个模块的功能产生不利的影响；

- 各个子功能组合起来，能否达到预期要求的父功能；
- 全局数据结构是否有问题；
- 单个模块的误差累积起来，是否会放大，从而达到不能接受的程度。

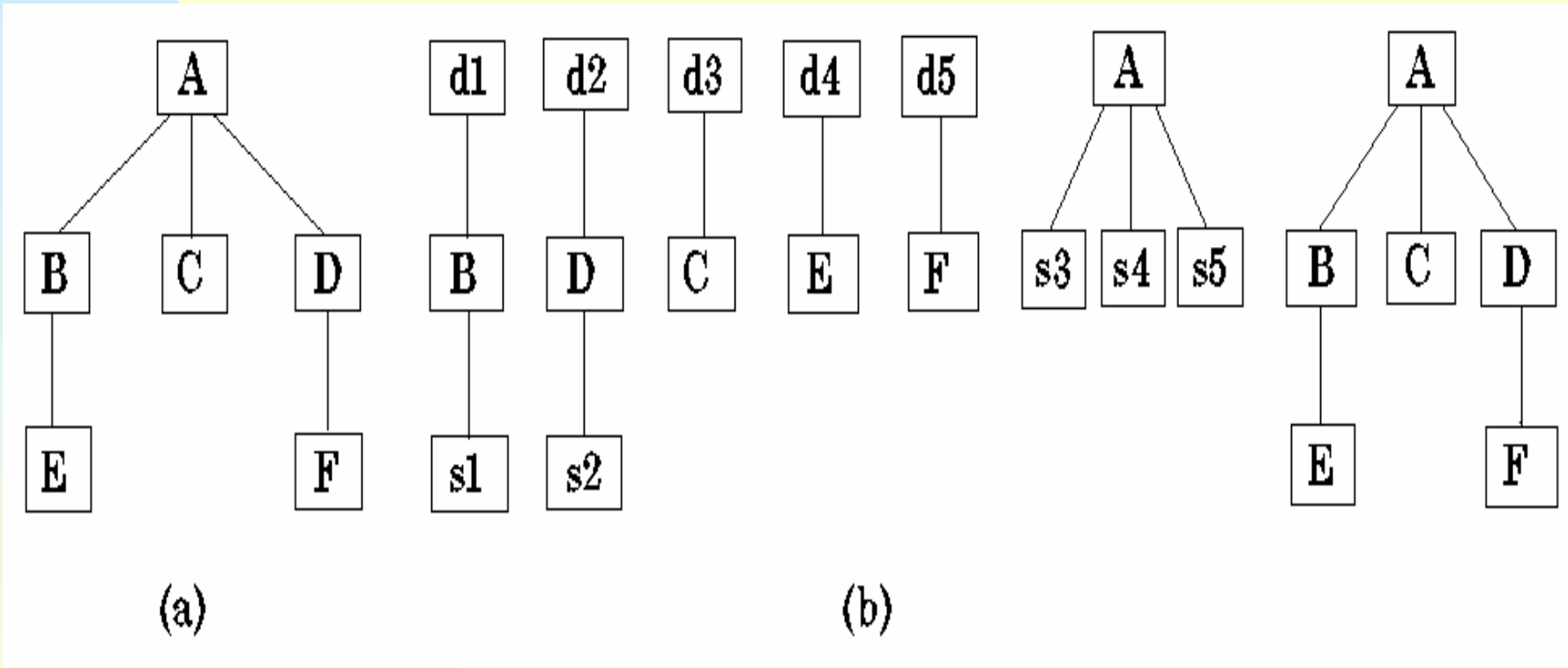
在单元测试的同时可进行组装测试，

发现并排除在模块连接中可能出现的问题，最终构成要求的软件系统。

- 子系统的组装测试特别称为部件测试，它所做的工作是要找出组装后的子系统与系统需求规格说明之间的不一致。
- 通常，把模块组装成为系统的方式有两种
 - 一次性组装方式
 - 增殖式组装方式

1. 一次性组装方式 (big bang)

- 它是一种非增殖式组装方式。也叫做整体拼装。
- 使用这种方式，首先对每个模块分别进行模块测试，然后再把所有模块组装在一起进行测试，最终得到要求的软件系统。

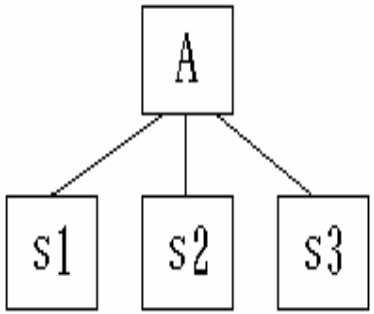

A
B
C
D
E
F
d1
d2
d3
d4
d5
B
D
C
E
F
s1
s2
A
s3
s4
s5
A
B
C
D
E
F
(a)
(b)

2. 增殖式组装方式

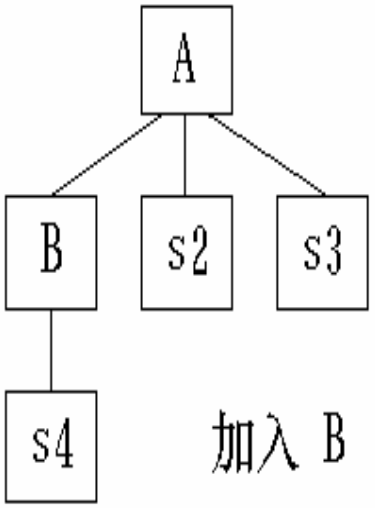
- 这种组装方式又称渐增式组装
- 首先对一个个模块进行模块测试，然后将这些模块逐步组装成较大的系统
- 在组装的过程中边连接边测试，以发现连接过程中产生的问题
- 通过增殖逐步组装成为要求的软件系统。

(1) 自顶向下的增殖方式

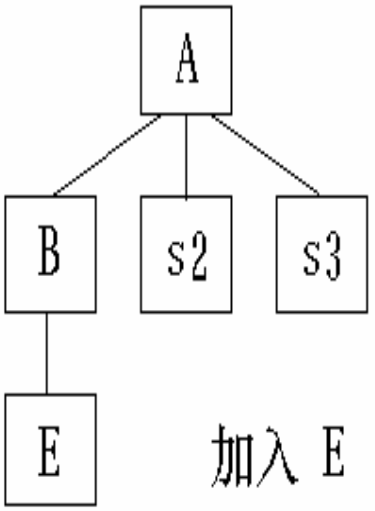
- 这种组装方式将模块按系统程序结构，沿控制层次自顶向下进行组装。
- 自顶向下的增殖方式在测试过程中较早地验证了主要的控制和判断点。
- 选用按深度方向组装的方式，可以首先实现和验证一个完整的软件功能。



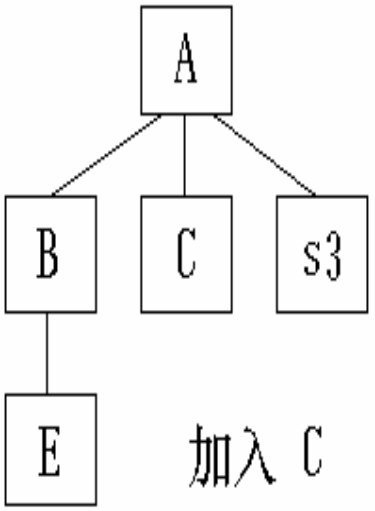
测试 A



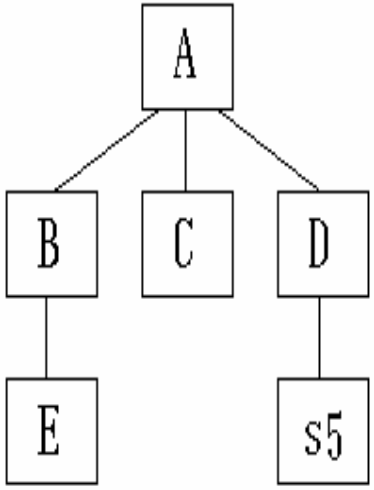
加入 B



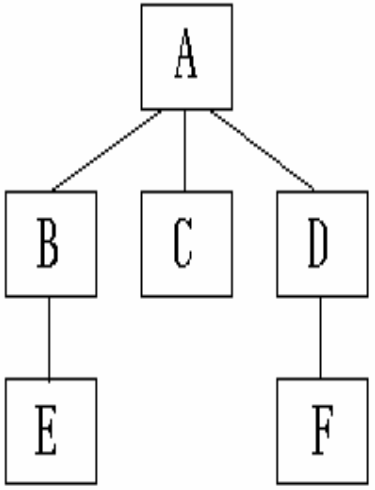
加入 E



加入 C



加入 D

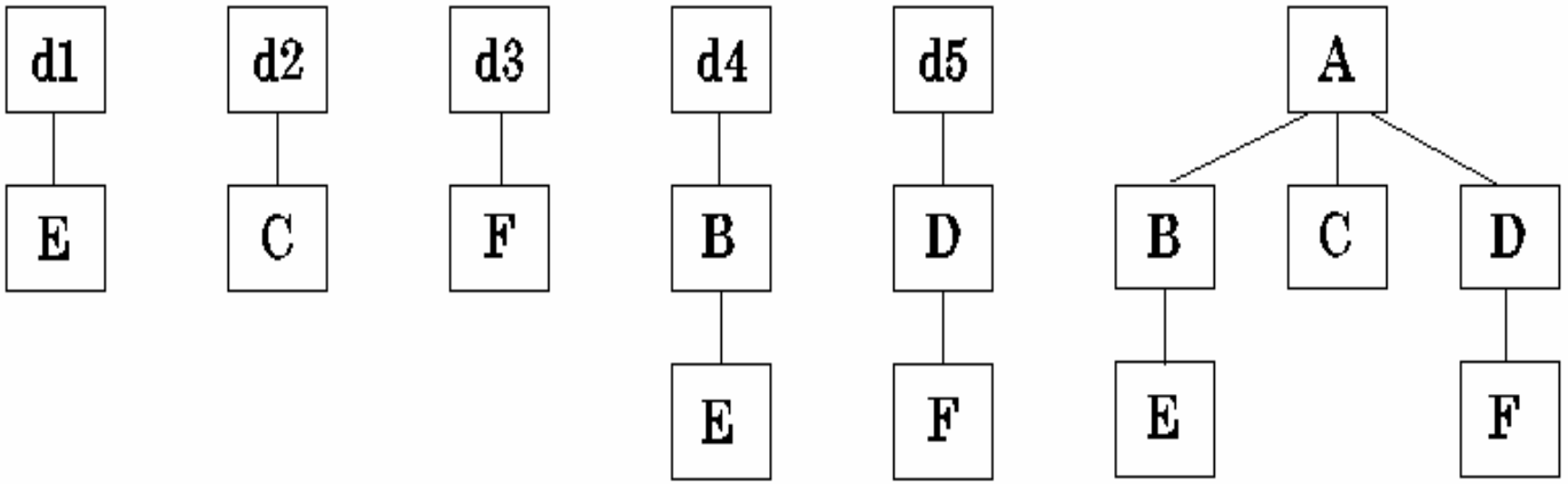


加入 F

按深度方向组装的例子

(2) 自底向上的增殖方式

- 这种组装的方式是从程序模块结构的最底层的模块开始组装和测试。
- 因为模块是自底向上进行组装，对于一个给定层次的模块，它的子模块（包括子模块的所有下属模块）已经组装并测试完成，所以不再需要桩模块。在模块的测试过程中需要从子模块得到的信息可以直接运行子模块得到。



- 自顶向下增殖的方式和自底向上增殖的方式各有优缺点。
- 一般来讲，一种方式的优点是另一种方式的缺点。

(3) 混合增殖式测试

- 衍变的自顶向下的增殖测试
 - 首先对输入／输出模块和引入新算法模块进行测试;
 - 再自底向上组装成为功能相当完整且相对独立的子系统;
 - 然后由主模块开始自顶向下进行增殖测试。

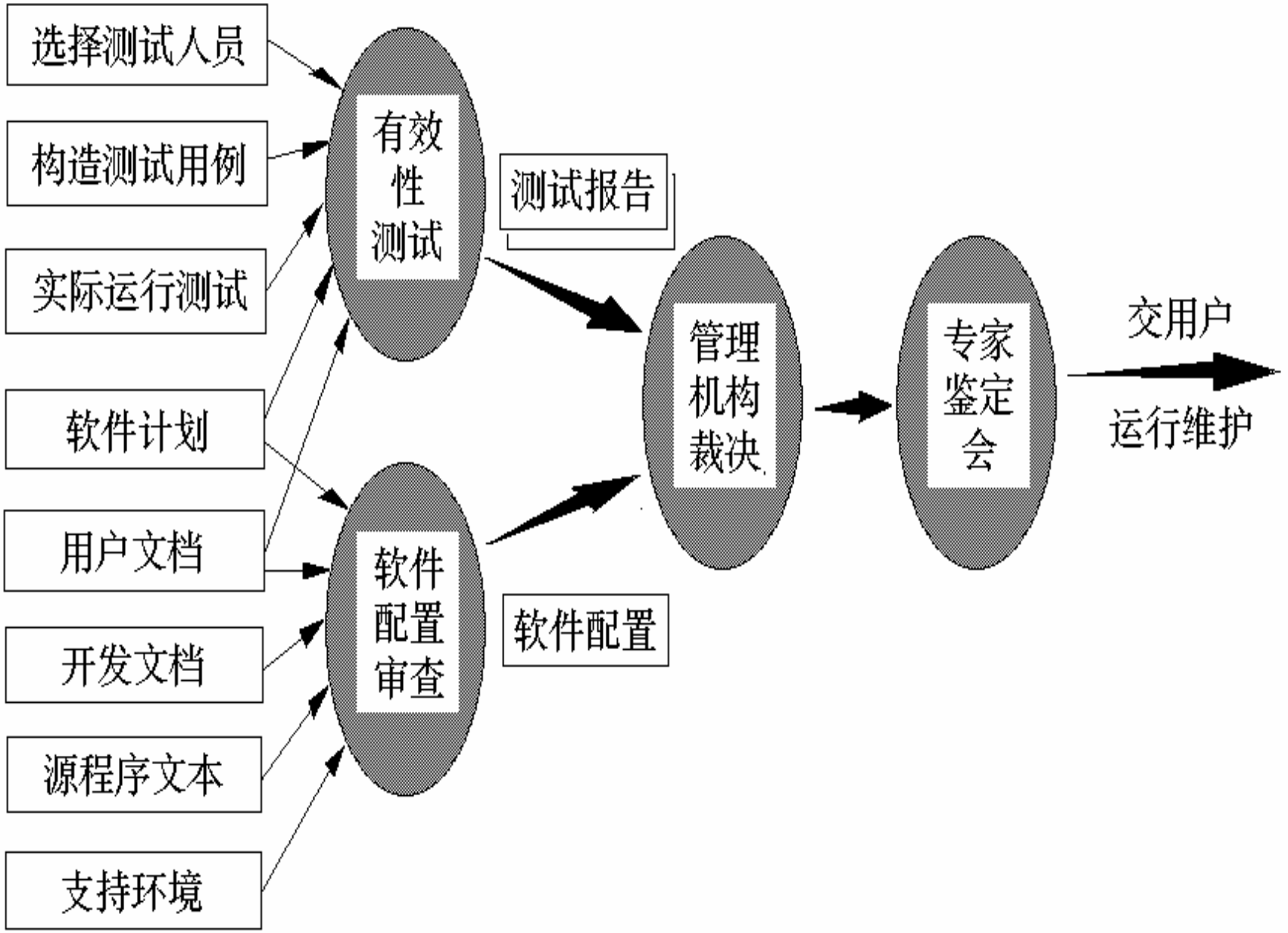
- 自底向上–自顶向下的增殖测试
 - 首先对含读操作的子系统自底向上直至根结点模块进行组装和测试;
 - 然后对含写操作的子系统做自顶向下的组装与测试。
- 回归测试
 - 这种方式采取自顶向下的方式测试被修改的模块及其子模块;
 - 然后将这一部分视为子系统，再自底向上测试。

关键模块问题

- 在组装测试时，应当确定关键模块，对这些关键模块及早进行测试。
- 关键模块的特征：
 ① 满足某些软件需求；
 ② 在程序的模块结构中位于较高的层次（高层控制模块）；
 ③ 较复杂、较易发生错误；
 ④ 有明确定义的性能要求。

确认测试（Validation Testing）

- 确认测试又称有效性测试。任务是验证软件的功能和性能及其它特性是否与用户的要求一致。
- 对软件的功能和性能要求在软件需求规格说明书中已经明确规定。它包含的信息就是软件确认测试的基础。


选择测试人员
构造测试用例
实际运行测试
软件计划
用户文档
开发文档
源程序文本
支持环境
有效性测试
软件配置审查
测试报告
软件配置
管理机构裁决
专家鉴定会
交用户
运行维护

1. 进行有效性测试（黑盒测试）

- 有效性测试是在模拟的环境 (可能就是开发的环境) 下，运用黑盒测试的方法，验证被测软件是否满足需求规格说明书列出的需求。
- 首先制定测试计划，规定要做测试的种类。还需要制定一组测试步骤，描述具体的测试用例。

- 通过实施预定的测试计划和测试步骤，确定
 - 软件的特性是否与需求相符；
 - 所有的文档都是正确且便于使用；
 - 同时，对其它软件需求，例如可移植性、兼容性、出错自动恢复、可维护性等，也都要进行测试

- 在全部软件测试的测试用例运行完后，所有的测试结果可以分为两类：
 - **测试结果与预期的结果相符**。这说明软件的这部分功能或性能特征与需求规格说明书相符合，从而这部分程序被接受。
 - **测试结果与预期的结果不符**。这说明软件的这部分功能或性能特征与需求规格说明不一致，因此要为它提交一份问题报告。

2. 软件配置复查

- 软件配置复查的目的是保证
 - 软件配置的所有成分都齐全；
 - 各方面的质量都符合要求；
 - 具有维护阶段所必需的细节；
 - 而且已经编排好分类的目录。
- 应当严格遵守用户手册和操作手册中规定的使用步骤，以便检查这些文档资料的完整性和正确性。

验收测试（Acceptance Testing）

- 在通过了系统的有效性测试及软件配置审查之后，就应开始系统的验收测试。
- 验收测试是以用户为主的测试。软件开发人员和QA（质量保证）人员也应参加。
- 由用户参加设计测试用例，使用生产中的实际数据进行测试。

- 在测试过程中，除了考虑软件的功能和性能外，还应对软件的可移植性、兼容性、可维护性、错误的恢复功能等进行确认。
- 确认测试应交付的文档有：
 - 确认测试分析报告
 - 最终的用户手册和操作手册
 - 项目开发总结报告。

系统测试（System Testing）

- 系统测试，是将通过确认测试的软件，作为整个基于计算机系统的一个元素，与计算机硬件、外设、某些支持软件、数据和人员等其它系统元素结合在一起，在实际运行环境下，对计算机系统进行一系列的组装测试和确认测试。
- 系统测试的目的在于通过与系统的需求定义作比较，发现软件与系统的定义不符合或与之矛盾的地方。

α测试和β测试

- 在软件交付使用之后，用户将如何实际使用程序，对于开发者来说是无法预测的。
- α测试是由一个用户在开发环境下进行的测试，也可以是公司内部的用户在模拟实际操作环境下进行的测试。

- α测试的目的是评价软件产品的FLURPS（即功能、局域化、可使用性、可靠性、性能和支持）。尤其注重产品的界面和特色。
- α测试可以从软件产品编码结束之时开始，或在模块（子系统）测试完成之后开始，也可以在确认测试过程中产品达到一定的稳定和可靠程度之后再开始。

- β测试是由软件的多个用户在实际使用环境下进行的测试。这些用户返回有关错误信息给开发者。
- 测试时，开发者通常不在测试现场。因而，β测试是在开发者无法控制的环境下进行的软件现场应用。
- 在β测试中，由用户记下遇到的所有问题，包括真实的以及主观认定的，定期向开发者报告。

- β测试主要衡量产品的FLURPS。着重于产品的支持性，包括文档、客户培训和支持产品生产能力。
- 只有当α测试达到一定的可靠程度时，才能开始β测试。它处在整个测试的最后阶段。同时，产品的所有手册文本也应该在此阶段完全定稿。



测试种类

- 软件测试是由一系列不同的测试组成。主要目的是对以计算机为基础的系统进行充分的测试。

功能测试

功能测试是在规定的一段时间内运行软件系统的所有功能，以验证这个软件系统有无严重错误。

可靠性测试

如果系统需求说明书中有对可靠性的要求，则需进行可靠性测试。

① 平均失效间隔时间 MTBF (Mean Time Between Failures) 是否超过规定时限?

② 因故障而停机的时间 MTTR (Mean Time To Repairs) 在一年中应不超过多少时间。

强度测试

强度测试是要检查在系统运行环境不正常乃至发生故障的情况下，系统可以运行到何种程度的测试。例如：

- 把输入数据速率提高一个数量级，确定输入功能将如何响应。
- 设计需要占用最大存储量或其它资源的测试用例进行测试。

- 设计出在虚拟存储管理机制中引起“颠簸”的测试用例进行测试。
- 设计出会对磁盘常驻内存的数据过度访问的测试用例进行测试。

- 强度测试的一个变种就是敏感性测试。在程序有效数据界限内一个小范围内的一组数据可能引起极端的或不平稳的错误处理出现，或者导致极度的性能下降的情况发生。此测试用以发现可能引起这种不稳定性或不正常处理的某些数据组合。

性能测试

- 性能测试是要检查系统是否满足在需求说明书中规定的性能。特别是对于实时系统或嵌入式系统。
- 性能测试常常需要与强度测试结合起来进行，并常常要求同时进行硬件和软件检测。
- 通常，对软件性能的检测表现在以下几个方面：响应时间、吞吐量、辅助存储区，例如缓冲区，工作区的大小等、处理精度，等等。

恢复测试

恢复测试是要证实在克服硬件故障(包括掉电、硬件或网络出错等)后，系统能否正常地继续进行工作，并不对系统造成任何损害。

- 为此，可采用各种人工干预的手段，模拟硬件故障，故意造成软件出错。并由此检查：
 - 错误探测功能——系统能否发现硬件失效与故障；

- 能否切换或启动备用的硬件；
- 在故障发生时能否保护正在运行的作业和系统状态；
- 在系统恢复后能否从最后记录下来的无错误状态开始继续执行作业，等等。
- 掉电测试：其目的是测试软件系统在发生电源中断时能否保护当时的状态且不毁坏数据，然后在电源恢复时从保留的断点处重新进行操作。

启动／停止测试

这类测试的目的是验证在机器启动及关机阶段，软件系统正确处理的能力。

这类测试包括

- 反复启动软件系统 (例如，操作系统自举、网络的启动、应用程序的调用等)
- 在尽可能多的情况下关机。

配置测试

- 这类测试是要检查计算机系统内各个设备或各种资源之间的相互联结和功能分配中的错误。
- 它主要包括以下几种：
 - 配置命令测试：验证全部配置命令的可操作性（有效性）；特别对最大配置和最小配置要进行测试。软件配置和硬件配置都要测试。

- **循环配置测试**：证明对每个设备物理与逻辑的，逻辑与功能的每次循环置换配置都能正常工作。
- **修复测试**：检查每种配置状态及哪个设备是坏的。并用自动的或手工的方式进行配置状态间的转换。

安全性测试

安全性测试是要检验在系统中已经存在的系统安全性、保密性措施是否发挥作用，有无漏洞。

- 力图破坏系统的保护机构以进入系统的主要方法有以下几种：
 - 正面攻击或从侧面、背面攻击系统中易受损坏的那些部分；
 - 以系统输入为突破口，利用输入的容错性进行正面攻击；

- 申请和占用过多的资源压垮系统，以破坏安全措施，从而进入系统；
- 故意使系统出错，利用系统恢复的过程，窃取用户口令及其它有用的信息；
- 通过浏览残留在计算机各种资源中的垃圾（无用信息），以获取如口令，安全码，译码关键字等信息；
- 浏览全局数据，期望从中找到进入系统的关键字；
- 浏览那些逻辑上不存在，但物理上还存在的各种记录和资料等。

可使用性测试

- 可使用性测试主要从使用的合理性和方便性等角度对软件系统进行检查，发现人为因素或使用上的问题。
- 要保证在足够详细的程度下，用户界面便于使用；对输入量可容错、响应时间和响应方式合理可行、输出信息有意义、正确并前后一致；出错信息能够引导用户去解决问题；软件文档全面、正规、确切。

可支持性测试

这类测试是要验证系统的支持策略对于公司与用户方面是否切实可行。

- 它所采用的方法是
 - 试运行支持过程(如对有错部分打补丁的过程，热线界面等)；
 - 对其结果进行质量分析；
 - 评审诊断工具；
 - 维护过程、内部维护文档；
 - 修复一个错误所需平均最少时间。

安装测试

安装测试的目的不是找软件错误，而是找安装错误。

- 在安装软件系统时，会有多种选择。
 - 要分配和装入文件与程序库
 - 布置适用的硬件配置
 - 进行程序的联结。
- 而安装测试就是要找出在这些安装过程中出现的错误。

- 安装测试是在系统安装之后进行测试。它要检验：
 - 用户选择的一套任选方案是否相容；
 - 系统的每一部分是否都齐全；
 - 所有文件是否都已产生并确有所需要的内容；
 - 硬件的配置是否合理，等等。

过程测试

- 在一些大型的系统中，部分工作由软件自动完成，其它工作则需由各种人员，包括操作员，数据库管理员，终端用户等，按一定规程同计算机配合，靠人工来完成。
- 指定由人工完成的过程也需经过仔细的检查，这就是所谓的过程测试。

互连测试

- 互连测试是要验证两个或多个不同的系统之间的互连性。

兼容性测试

- 这类测试主要想验证软件产品在不同版本之间的兼容性。有两类基本的兼容性测试：
 - 向下兼容
 - 交错兼容

容量测试

- 容量测试是要检验系统的能力最高能达到什么程度。例如，
 - 对于编译程序，让它处理特别长的源程序；
 - 对于操作系统，让它的作业队列“满员”；
 - 对于信息检索系统，让它使用频率达到最大。

在使系统的全部资源达到“满负荷”的情形下，测试系统的承受能力。

文档测试

这种测试是检查用户文档(如用户手册)的清晰性和精确性。

- 用户文档中所使用的例子必须在测试中一一试过，确保叙述正确无误。



调试（Debug）

- 软件调试是在进行了成功的测试之后才开始的工作。它与软件测试不同，调试的任务是进一步诊断和改正程序中潜在的错误。
- 调试活动由两部分组成：
 - 确定程序中可疑错误的确切性质和位置。
 - 对程序(设计,编码)进行修改，排除这个错误。

- 调试工作是一个具有很强技巧性的工作。
- 软件运行失效或出现问题，往往只是潜在错误的外部表现，而外部表现与内在原因之间常常没有明显的联系。如果要找出真正的原因，排除潜在的错误，不是一件易事。
- 可以说，调试是通过现象，找出原因的一个思维分析的过程。

调试的步骤

(1) 从错误的外部表现形式入手，确定程序中出错位置；

(2) 研究有关部分的程序，找出错误的内在原因；

(3) 修改设计和代码，以排除这个错误；

(4) 重复进行暴露了这个错误的原始测试或某些有关测试。

- 从技术角度来看，查找错误的难度在于：
 - 现象与原因所处的位置可能相距甚远。
 - 当其它错误得到纠正时，这一错误所表现出的现象可能会暂时消失，但并未实际排除。
 - 现象实际上是由一些非错误原因(例如，舍入不精确)引起的。

- 现象可能是由于一些不容易发现的人为错误引起的。
- 错误是由于时序问题引起的，与处理过程无关。
- 现象是由于难于精确再现的输入状态（例如，实时应用中输入顺序不确定）引起。
- 现象可能是周期出现的。在软、硬件结合的嵌入式系统中常常遇到。

几种主要的调试方法

调试的关键在于推断程序内部的错误位置及原因。可以采用以下方法：

强行排错

这种调试方法目前使用较多，效率较低。它不需要过多的思考，比较省脑筋。例如：

- 通过内存全部打印来调试，在这大量的数据中寻找出错的位置。

- 在程序特定部位设置打印语句，把打印语句插在出错的源程序的各个关键变量改变部位、重要分支部位、子程序调用部位，跟踪程序的执行，监视重要变量的变化。
- 自动调试工具。利用某些程序语言的调试功能或专门的交互式调试工具，分析程序的动态过程，而不必修改程序。

应用以上任一种方法之前，都应当对错误的征兆进行全面彻底的分析，得出对出错位置及错误性质的推测，再使用一种适当的调试方法来检验推测的正确性。

回溯法调试

这是在小程序中常用的一种有效的调试方法。

一旦发现了错误，人们先分析错误征兆，确定最先发现“症状”的位置。

然后，人工沿程序的控制流程，向回追踪源程序代码，直到找到错误根源或确定错误产生的范围。

- 例如，程序中发现错误处是某个打印语句。通过输出值可推断程序在这一点上变量的值。再从这一点出发，回溯程序的执行过程，反复考虑：“如果程序在这一点上的状态（变量的值）是这样，那么程序在上一点的状态一定是这样…”，直到找到错误的位置。

归纳法调试

- 归纳法是一种从特殊推断一般的系统化思考方法。归纳法调试的基本思想是：从一些线索(错误征兆)着手，通过分析它们之间的关系来找出错误。
 - 收集有关的数据　列出所有已知的测试用例和程序执行结果。看哪些输入数据的运行结果是正确的，哪些输入数据的运行结果有错误。

◆ 组织数据

由于归纳法是从特殊到一般的推断过程，所以需要组织整理数据，以发现规律。

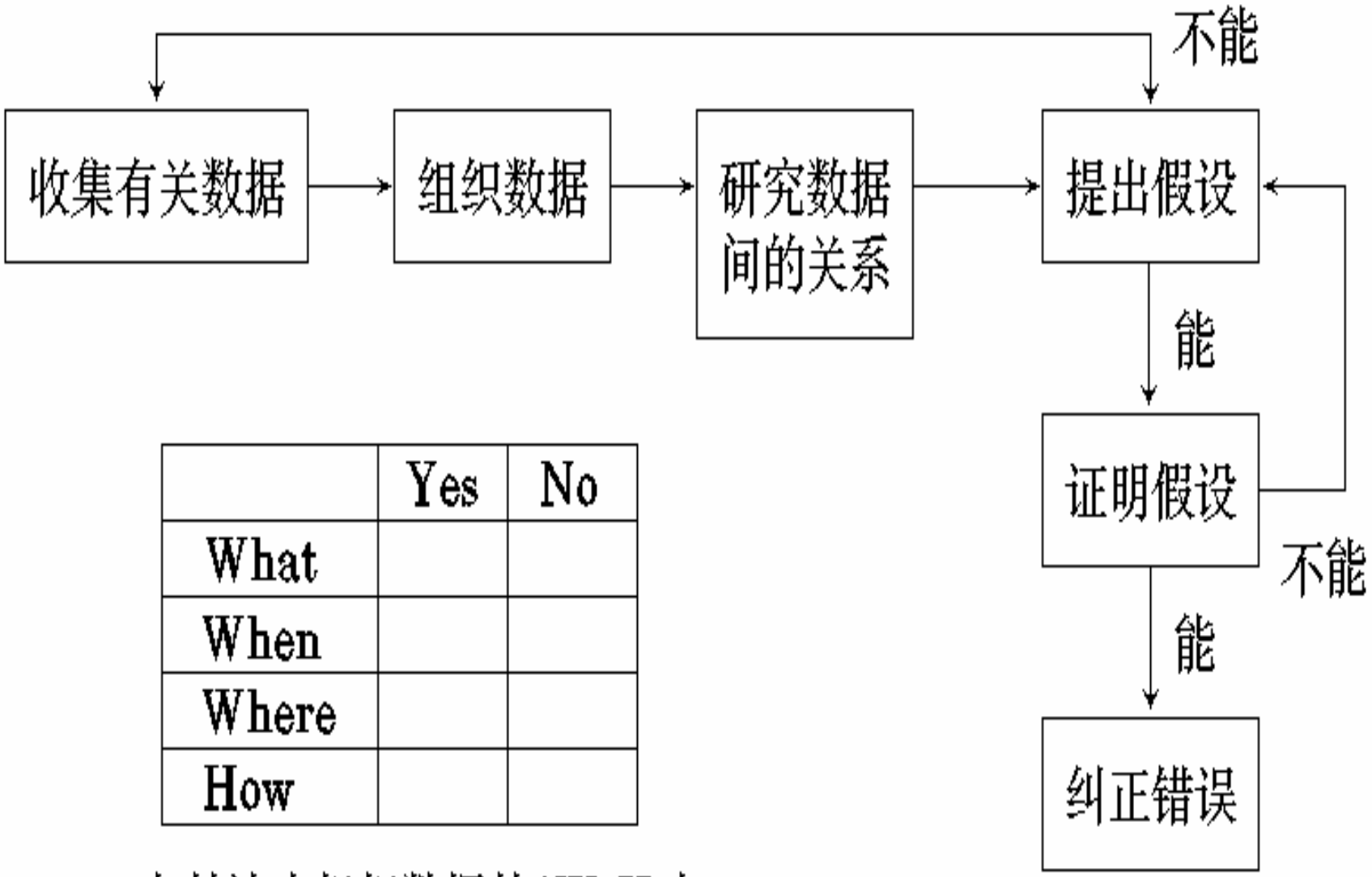
常以**3W1H**形式组织可用的数据：

“What” 列出一般现象；

“Where”说明发现现象的地点；

“When” 列出现象发生时所有已知情况；

“How” 说明现象的范围和量级；



	Yes	No
What		
When		
Where		
How		

归纳法中组织数据的3W1H表

“Yes”描述出现错误的3W1H；

“No”作为比较，描述了没有错误的3W1H。通过分析找出矛盾来。

- 提出假设

 分析线索之间的关系，利用在线索结构中观察到的矛盾现象，设计一个或多个关于出错原因的假设。如果一个假设也提不出来，归纳过程就需要收集更多的数据。此时，应当再设计与执行一些测试用例，以获得更多的数据。

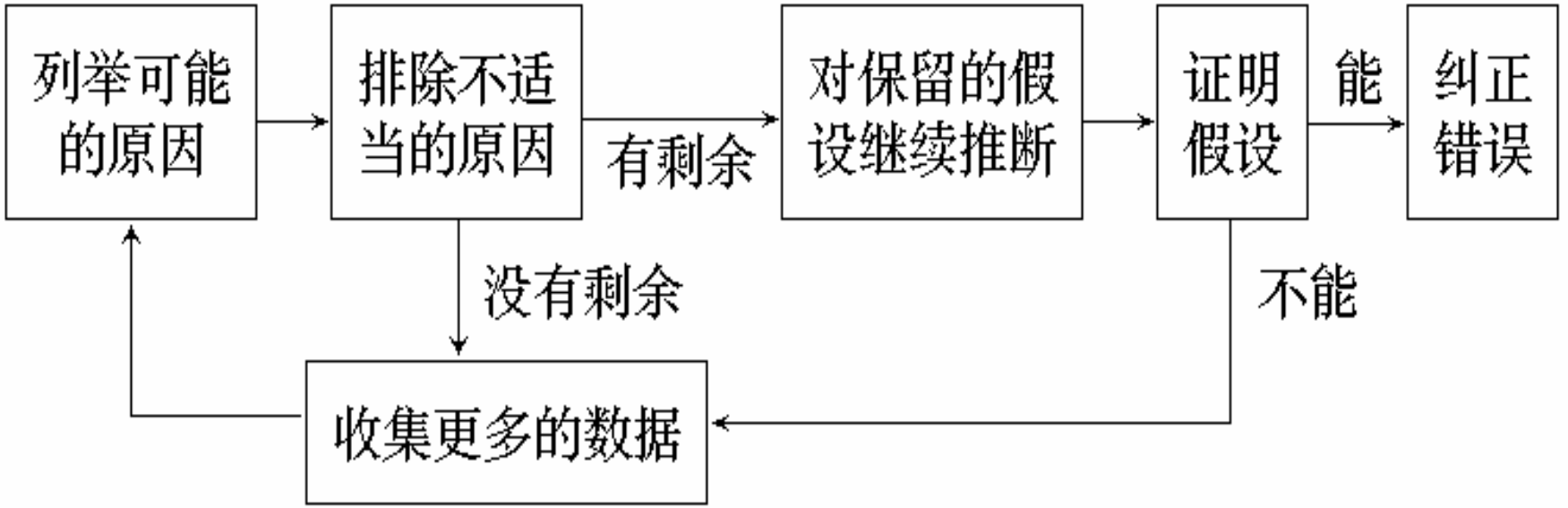
- **证明假设**

把假设与原始线索或数据进行比较，若它能完全解释一切现象，则假设得到证明；否则，就认为假设不合理，或不完全，或是存在多个错误，以致只能消除部分错误。

- **演绎法调试**

 演绎法是一种从一般原理或前提出发，经过排除和精化的过程来推导出结论的思考方法。演绎法排错是测试人员首先根据已有的测试用例，设想及枚举出所有可能出错的原因做为假设；然后再用原始测试数据或新的测试，从中逐个排除不可能正确的假设；最后，再用测试数据验证余下的假设确是出错的原因。

- **列举所有可能出错原因的假设**
 把所有可能的错误原因列成表。通过它们，可以组织、分析现有数据。
- **利用已有的测试数据，排除不正确的假设**
 仔细分析已有的数据，寻找矛盾，力求排除前一步列出所有原因。如果所有原因都被排除了，则需要补充一些数据(测试用例)，以建立新的假设。



- 改进余下的假设

 利用已知的线索，进一步改进余下的假设，使之更具体化，以便可以精确地确定出错位置。
- 证明余下的假设

调试原则

- 在调试方面，许多原则本质上是心理学方面的问题。调试由两部分组成，调试原则也分成两组。
- 确定错误的性质和位置的原则
 - 用头脑去分析思考与错误征兆有关的信息。
 - 避开死胡同。

- 只把调试工具当做辅助手段来使用。利用调试工具，可以帮助思考，但不能代替思考。
- 避免用试探法，最多只能把它当做最后手段。

- 修改错误的原则
 - 在出现错误的地方，很可能还有别的错误。

- 修改错误的一个常见失误是只修改了这个错误的征兆或这个错误的表现，而没有修改错误的本身。
- 当心修正一个错误的同时有可能会引入新的错误。
- 修改错误的过程将迫使人们暂时回到程序设计阶段。
- 修改源代码程序，不要改变目标代码。

